The Dissertation Committee for Yige Hu
certifies that this is the approved version of the following dissertation:

File system designs on low-latency storage devices

Committee:

Emmett Witchel, Supervisor

Vijay Chidambaram, Co-Supervisor

Atul Adya

Simon Peter

File system designs on low-latency storage devices

by

Yige Hu

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2020

To everyone

Acknowledgments

I would like to express my sincere gratitude to my advisor Emmett Witchel. He introduced me to the area of computer systems and taught me a great deal of lessons on how to become a system researcher. As an advisor, he is extremely supportive and always willing to help whenever a student gets into trouble. His rigorous attitude towards research and experiments has influenced me immensely.

I would like to thank my other advisor, Vijay Chidambaram, for detailed advice about my research on file and storage systems. His paper writing style and creativity in research idea generation have always been enlightening and inspiring.

I would specially thank Texas Advanced Computing Center (TACC) and the staff, Lei Huang, Tommy Minyard and John Cazes, who offered precious help and technical support.

Besides my advisors, I would like to thank the rest of my thesis committee members, Simon Peter and Atul Adya, for giving helpful comments on my thesis.

I would like to thank the former members of our research group who offered me invaluable help and advice during my early PhD years: Sangman Kim, Michael Lee, Yuanzhong Xu and Youngjin Kwon.

I would like to thank other collaborators who worked with me on TxFS: Zhiting Zhu, Ian Neal and Tianyu Cheng. And many other collaborators on the projects where I learned a lot: Mark Silberstein, Seonggu Huh and Tyler Hunt.

Throughout my years in graduate school, I am grateful to receive help and support from many other friends and fellow graduate students: Yuming Sheng, Youer Pu, Yingchen Wang, Yu Feng, Xinyu Wang, Yun Wu, Hangchen Yu, Zhipeng Jia, Trinabh Gupta, Sekwon Lee, Zuocheng Ren, Chunzhi Su and Chao Xie.

Finally, I would like to thank my family for their love and encouragement. Thank my parents for creating such an open and supportive family environment for me to grow up.

YIGE HU

The University of Texas at Austin

August, 2020

File system designs on low-latency storage devices

Yige Hu, Ph.D.
The University of Texas at Austin, 2020

Supervisors: Emmett Witchel
Vijay Chidambaram

The evolution of modern applications and storage technologies has created new challenges for file systems. Applications store persistent state across multiple files and storage abstractions which must maintain crash consistency. New storage devices such as NAND flash memory has lower IO latency and higher bandwidth compared to spinning disks, especially for random memory accesses. Non-volatile memory (NVM) further decreases read and write latency for persistent storage, with access times on the order of DRAM. Decreased device latencies makes it more important to eliminate CPU overhead and system call overhead than to reduce accesses to the storage device.

We first introduce TxFS, a transactional file system that builds general-purpose transactions upon an existing file system's atomic-update mechanism such as journaling. Though prior work has explored a number of transactional file systems, TxFS has a unique set of properties: a simple API, portability across different hardware, high performance, low complexity (by building on the file-system journal), and full ACID transactions. We port SQLite, OpenLDAP, and Git to use

TxFS, and experimentally show that TxFS provides strong crash consistency while providing equal or better performance.

We then introduce NVMKVFS, a user-space file-system prototype focusing on metadata operations. We examine ext4-dax and NOVA scalability, finding numerous bottlenecks in the VFS. NVM's low memory access latency makes it more important to eliminate CPU overhead than to reduce accesses to the NVM device, so we eliminate the VFS caches while maintaining performance on standard workloads and increasing scalability. We propose a file system based on two global indexes on NVM, one for metadata and the other for data. Because of NVM's asymmetric bandwidth on read and write accesses, We focus on optimizing read-heavy workloads and metadata-heavy workloads in which write sizes are small.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

File systems provide abstractions for data stored on physical media; maintaining metadata structures and providing methods to operate on the data and metadata. File systems must provide certain guarantees to applications, including crash consistency. Crash consistency requires a file system to recover to a consistent state after a crash. Consistency can be a complex invariant that must hold across different data structures, and sometimes across different files. For example, file creation requires consistency between the metadata of the created file and the data blocks of its parent directory that link the newly created file into the file system's name space.

The evolution of application requirements and hardware capabilities introduces new challenges as well as optimization opportunities for the problem of file-system crash-consistency. Many application-level storage systems, including embedded databases and key-value stores, are built on top of file systems. Regardless of the crash consistency provided by the file system, they are required to provide application-specific guarantees, such as atomicity and isolation. Many applications store data across multiple abstractions. For example, Android's default mail application uses both file system APIs and queries to a SQLite [SQL] database. It implements an ad hoc application-level protocol to maintain consistency across the files that store an email's attachments and the SQLite database, which stores meta-

data about the attachments.

Low-latency storage devices such as non-volatile memory (NVM) introduce new challenges to current file system designs, while bringing opportunities for the file systems to optimize metadata indexing and multicore scalability. We run experiments on existing file systems with NVM support, including ext4 and NOVA [XS16], and expose several scalability bottlenecks in metadata indexing due to locks in the Virtual File System (VFS). The VFS provides useful services like system call abstraction, metadata caches and concurrency control, but for NVM devices, the VFS increases the metadata lookup latency and decreases the scalability of the file system. The VFS caches can decrease the overhead of slow IO devices, like block-oriented hard disk drives (HDDs) and solid-state drives (SSDs). But for modern, low-latency, byte-addressable persistent memory, the CPU overhead incurred by the VFS can be more important than the device latencies.

The rest of this section introduces two works. TxFS is a transactional file system built on top of the crash consistency mechanisms of existing file systems. It provides user applications with stronger crash semantics, better performance and easy-to-use APIs. NVMKVFS is a user-space file system prototype for demonstrating a scalable data structure design whose goal is to improve file system metadata performance on persistent memory devices.

1.1 From crash consistency to transactions

Modern applications store persistent state across multiple files [PCA+14], sometimes splitting their state among embedded databases, key-value stores, and

file systems [SBL+14]. Such applications need to ensure that their data is not corrupted or lost in the event of a crash. Unfortunately, existing techniques for crash consistency result in complex protocols and subtle bugs [PCA+14]. For example, Android's default mail application stores messages in a SQLite database and attachments as files. The database also keeps pointers to the attachment files. Without crash consistency protocols, a crash between file and database updates can lead to a dangling pointer where the file name of an attachment is valid, but the file itself is missing. To resolve the issue, the mail application uses a set of carefully arranged `fsync()` calls, leading to low performance and an error-prone implementation.

Transactions present an intuitive way to atomically update persistent state [Gra81]. Unfortunately, building transactional systems is complex and error-prone. The first part of this proposal introduces a novel approach to building a transactional file system. We take advantage of a mature, well-tested piece of functionality in the operating system: the file-system journal, which is used to ensure atomic updates to the internal state of the file system. We use the atomicity and durability provided by journal transactions and leverage it to build user-space ACID transactions. Our approach greatly reduces the development effort and complexity involved in building a transactional file system.

We propose TxFS: a transactional file system that builds on the ext4 file system's journaling mechanism. We designed TxFS to be practical to implement and to use. TxFS has a unique set of properties: it has a small implementation (5,200 LOC) by building on the journal (for example, TxFS has 25% the LOC of the TxOS transactional operating system [PHR+09]); it provides high

performance unlike various solutions which built a transactional file system using a user-space database [Geh94, Ols93, MTV02, Wri07]; it has a simple API (just wrap code in `fs_tx_begin()` and `fs_tx_commit()`) compared to solutions like Valor [SGC+09] or TxF [Rus05] which require multiple system calls per transaction and can require the developer to understand implementation details like logging; it provides all ACID guarantees unlike solutions such as CFS [Min15] and AdvFS [VMP+15] which only offer some of the guarantees; it provides transactions at the file level instead of at the block level unlike Isotope [Shi16], making several optimizations easier to implement; finally, TxFS does not depend upon specific properties of the underlying storage unlike solutions such as MARS [Cob13] and TxFlash [PRZ08]. We test TxFS on SSDs and show performance boost of $1.6\times$ on the SQLite TPC-C benchmark, and up to $12.5\times$ on the LDIF backend of the Berkeley DB (BDB) with little porting effort.

1.2 Fast metadata indexes for NVM file system

Low-latency storage devices such as non-volatile memory (NVM) challenge current file system designs, requiring a rethink of metadata indexing and multicore scalability. File systems in the Linux kernel rely on the VFS layer for caching and concurrency control. Caching in the VFS is helpful for slow storage devices. But a recent study shows that it is less efficient on persistent memory (PMEM) with low latency, high throughput and byte-addressability[WJX18, VNP+14]. On the other hand, the concurrency control mechanism in the VFS layer, including locks such as the global hash lock and inode locks, limits file system scalability when run-

ning metadata-heavy workloads. NVM-optimized file systems such as Ext4-dax, XFS-dax and NOVA rely on the VFS layer for caching and concurrency control. As a result, their scalability is bottlenecked by VFS synchronization. Other research [WJX18, VNP+14] shows that the performance overhead in the VFS layer is more significant on persistent memory than on traditional block-based devices. Lower device latency removes the bottleneck caused by IO overhead. CPU overhead, such as indexing and synchronization, has become the new bottleneck.

We design a set of scalable data structures as key components for future NVM file system design, and build NVMKVFS, a user-space file system prototype that supports POSIX APIs, to demonstrate the throughput and scalability improvement brought by the data structures. We prioritize scalable multicore performance in the prototype file system design. The low memory access latency of persistent memory makes it possible to eliminate the VFS caches while maintaining performance. We build NVMKVFS on top of two global indexes on NVM, one for metadata and the other for data indexing. By implementing file system operations with the APIs provided by the underlying persistent and highly parallel indexes, NVMKVFS maximizes concurrency by removing the scalability bottlenecks caused by locking in the VFS layer. Because NVM's read bandwidth can be nearly two times higher than its write bandwidth[IYZ+19], we focus on optimizing read-heavy workloads and metadata-heavy workloads in which write sizes are small. We test NVMKVFS on both Intel Optane DC persistent memory and DRAM-emulated PMEM. On PMEM hardware, when running file creation workload in a shared directory, NVMKVFS gets a $7.45\times$ speedup over 28 cores compared to ext4-dax's

$1.23\times$ speedup.

Chapter 2

Background and motivation

Crash consistency is an important guarantee when people design storage systems. A system is defined as crash-consistent if it can recover to a consistent state after a system crash or a power failure. File systems maintain metadata for the information on the whole file system, extra information on files and structural information such as data page indexes. In order to maintain crash consistency, such as the integrity and consistency of the superblocks, inode and data block allocation bitmaps, inodes and data page indexing structures, file systems use mechanisms such as logging and copy-on-write (COW). User space applications built on file systems further introduce needs for the crash consistency across persistent state, sometimes in different files or even different storage abstractions.

Persistent memory is a new type of storage hardware that brings novel challenge and opportunities to novel file system design. It provides data persistence and affordable large capacity with throughput and latency on the level of DRAM. To take full advantage of the hardware's performance, NVM file systems need to factor PMEM's memory characteristics and byte-addressability into their design.

This chapter discusses how modern applications provide crash consistency, how file systems index the metadata in a crash-consistent way and how the emergence of the persistent memory is potentially affecting modern file system design.

2.1 Application-level crash consistency

We first describe the protocols used by current applications to update state in a crash-consistent manner. We then present a study of different applications and the challenges they face in maintaining crash consistency across persistent state stored in different abstractions. We describe the file-system optimizations enabled by transactions and finally summarize why we think transactional file systems should be revisited.

2.1.1 How applications update state today

Given that applications today do not have access to transactions, how do they consistently update state to multiple storage locations? Even if the system crashes or power fails, applications need to maintain invariants across state in different files (*e.g.,* an image file should match the thumbnail in a picture gallery). Applications achieve this by using ad hoc protocols that are complex and error-prone [PCA+14].

In this section, we show how difficult it is to implement seemingly simple protocols for consistent updates to storage. There are many details that are often overlooked, like the persistence of directory contents. These protocols are complex, error prone, and inefficient. With current storage technologies, these protocols must sacrifice performance to be correct because there is no efficient way to order storage updates.

Currently, applications use the `fsync()` system call to order updates to storage [CPADAD13]; since `fsync()` forces durability of data, the latency of

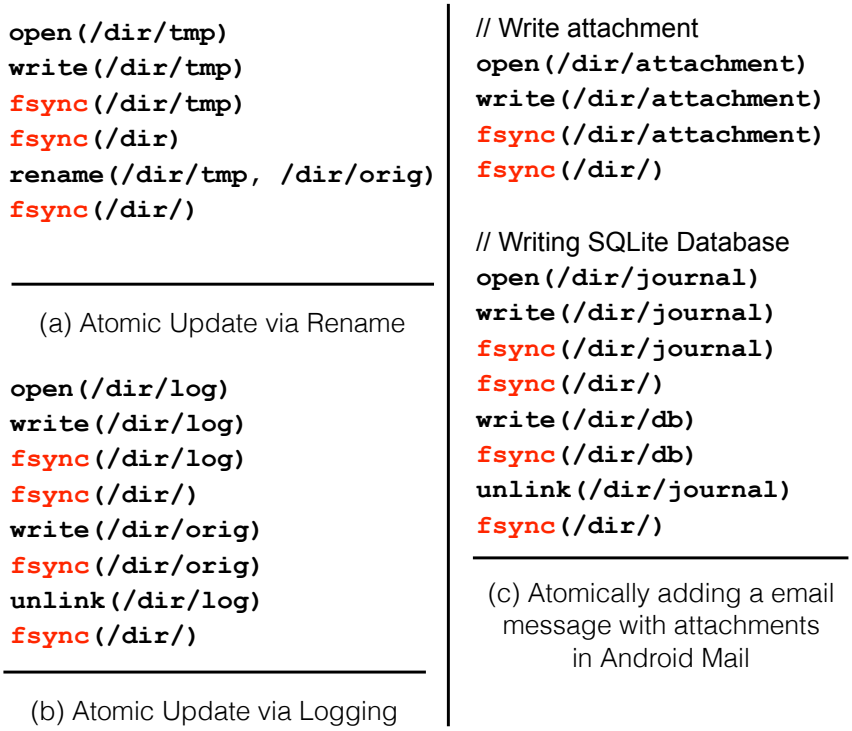

Figure 2.1: Different protocols used by applications to make consistent updates to persistent data.

a `fsync()` call varies from a few milliseconds to several seconds. As a result, applications do not call `fsync()` at all the places in the update protocol where it is necessary, leading to severe data loss and corruption bugs [PCA+14].

We now describe two common techniques used by applications to consistently update stable storage. Figure 2.1 illustrates these protocols.

Atomic rename. Protocol (a) shows how a file can be updated via atomic rename. The atomic rename approach is widely used by editors, such as Emacs and Vim, and by GNOME applications that need to atomically update dot configuration files.

The application writes new data to a temporary file, persists it with an `fsync()` call, updates the parent directory with another `fsync()` call, and then renames the temporary file *over* the original file, effectively causing the directory entry of the original file to point to the temporary file instead. The old contents of the original file are unlinked and deleted. Finally, to ensure that the temporary file has been unlinked properly, the application calls `fsync()` on the parent directory.

Logging. Protocol (b) shows another popular technique for atomic updates, logging [Hag87] (either write-ahead-logging or undo logging). The log file is written with new contents, and both the log file and the parent directory (with the new pointer to log file) are persisted. The application then updates the original file and persists the original file; the parent directory does not change during this step. Finally, the log is unlinked, and the parent directory is persisted.

The situation becomes more complex when applications store state across multiple files. Protocol (c) illustrates how the Android Mail application adds a new email with an attachment. The attachment is stored on the file system, while the email message (along with metadata) is stored in the database (which for SQLite, also resides on the file system). Since the database has a pointer to the attachment (i.e., a file name), the attachment must be persisted first. Persisting the attachment requires two `fsync()` calls (to the file and its containing directory) [PCA+14, fsy]. SQLite's most performant mode uses write-ahead-logging to atomically update the database. It then follows a protocol similar to Protocol (b).

Removing `fsync()` calls in any of the presented protocols will lead to data loss or corruption. For instance, in Protocol (b), if the parent directory is not

persisted with an `fsync()` call, the following scenario could occur: the application writes the log file, and then starts overwriting the original file in place. The system crashes at this point. Upon reboot, the log file does not exist, since the directory entry pointing to the log file was not persisted. Thus, the application file has been irreversibly partially edited, and cannot be restored to a consistent version. Many application developers avoid `fsync()` calls due to the resulting decrease in performance, leading to severe bugs that cause loss of data.

Safe update protocols for stable storage are complex and low performance (*e.g.,* Android Mail uses *six* `fsync()` calls to persist a single email with an attachment). System support for transactions will provide high performance for these applications.

Double journaling. Figure 2.2 presents an example of how application ad hoc protocols reduce performance. SQLite uses write-ahead logging to provide atomic updates. To implement write-ahead logging, SQLite needs two synchronous writes: one to write data to the log and one to write data in-place to the file[1]. SQLite runs on top of a file system such as ext4, which also implements write-ahead logging for metadata updates. As a result, updates to the SQLite write-ahead log are journaled by the file system (metadata in ordered mode, both data and metadata in journal mode). This leads to extra writes and cache flushes to storage. This is termed the *double journaling problem*: maintaining an application-level journal suffers extra writes to the file system's journal. If the transaction is implemented inside the file

[1]With batched checkpointing, the second write can be done in the background

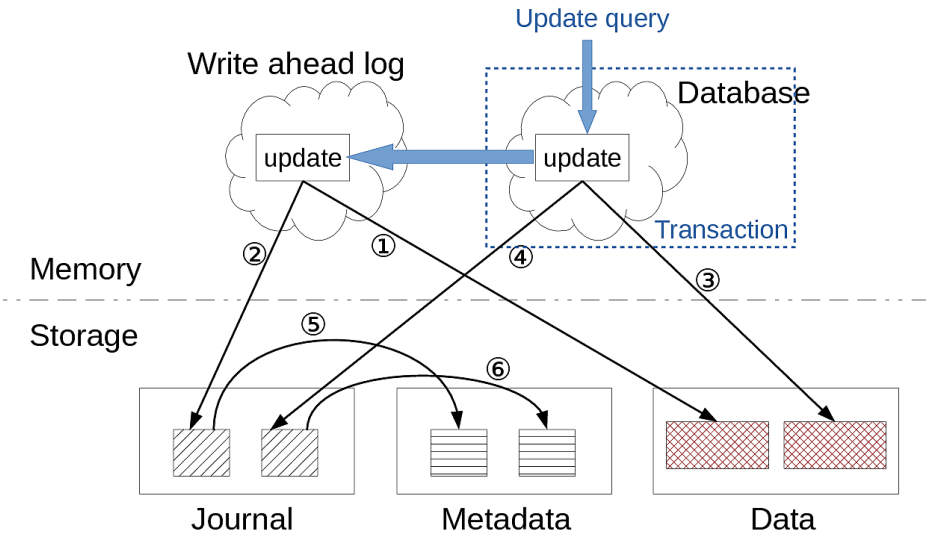

Figure 2.2: SQLite's double-journaling effect on ext4 with ordered journal mode. In ordered mode, data writes precede metadata writes. Block writes happen in the followed sequence. (1) Data write in write ahead log (WAL); (2) Metadata journal for the WAL write; (3) Data write in database file; (4) Metadata journal for the database write; (5) Asynchronous write back for the metadata for WAL; (6) Asynchronous write back for the metadata for database file. Data that resides in memory when asynchronously written back from the journal is written from memory (not read from the journal).

system, only two synchronous writes are required.

2.1.2 Application case studies

We present four examples of applications that struggle with obtaining crash consistency using primitives available today. Several applications store data across the file system, key-value stores, and embedded databases such as SQLite [SBL+14]. While all of this data ultimately resides in the file system, their APIs and performance constraints are different and consistently updating state across these systems

is complex and error-prone.

Android mail. Android's default mail application stores mail messages using the SQLite embedded database [SQL]. Mail attachments are stored separately as a file, and the database stores a pointer to the file. The user requires both the file and the database to be updated atomically; SQLite only ensures the database is updated correctly. For example, a crash could leave the database consistent, but with a dangling pointer to a missing attachment file. The mail application handles this by first persisting the attachment and the directory file which contains the attachment (via `fsync()`), and then persisting a database transaction. A transaction that spans both the database and the file system has a single commit point, simplifying semantics and possibly simplifying recovery. A single transaction would also eliminate the `fsync()` for the attachment, which can increase performance.

Mobile Photosharing Applications. Android Gallery stores metadata about photos (including a thumbnail) in a SQLite database while storing the files themselves directly on the file system. Thus a crash could leave the database updated with file information and a thumbnail (so it looks as if the file is there), but when the file is accessed, the user gets an error.

Apple iWork and iLife. Analysis of the storage behavior of Apple's home-user, desktop applications [HDV+12] finds that applications use a combination of the file system, key-value stores, and SQLite to store data. iTunes uses SQLite to store metadata similar to the Android Mail application. When you download a new song via iTunes, the sound file is transferred and the database updated with the

song's metadata. Apple's Pages application uses a combination of SQLite and key-value stores for user preferences and other metadata (two SQLite databases and 128 `.plist` key-value store files). Similar to Android Mail, Apple uses `fsync()` to order updates correctly.

Browsers. Mozilla Firefox stores user data in multiple SQLite databases. For example, addons, cookies, and download history are each stored in their separate SQLite database. Since downloads and other files are stored on the file system, a crash could leave a database with a dangling pointer to a missing file.

Synchronization services. Dropbox, Seafile, and Box allow users to keep their files synchronized across desktop and mobile systems and the cloud. The desktop clients for both services use SQLite to store file metadata such as when the file was last modified and synchronized. The desktop client of Box uses a key-value store in addition to SQLite and the file system. When a file is added to a synchronized folder, all these storage abstractions need to be updated atomically.

Version control systems. Git and Mercurial are widely-used version control systems. The `git commit` command requires two file-system operations to be atomic: a file append (`logs/HEAD`) and a file rename (to a lock file). Failure to achieve atomicity results in data loss and a corrupted repository [PCA+14]. Mercurial uses a combination of different files (`journal, filelog, manifest`) to consistently update state. Mercurial's `commit` requires a long sequence of file-system operations including file creations, appends, and renames be atomic; if not, the repository is corrupted [PCA+14].

For these applications, transactional support would lead directly to more understandable and more efficient idioms. It is difficult for a user-level program to provide crash-consistent transactional updates using the POSIX file-system interface. A transactional file-system interface will also enable high-performance idioms like editors grouping updates into transactions rather than the less efficient process they currently use of making temporary file copies that are committed via `rename`.

Note that applications techniques like atomic rename over temporary files do achieve crash consistency; however, a crash may lead to temporary files which need to be cleaned up. After a crash, the application runs a recovery procedure and returns to a consistent state. Often, the "recovery procedure" forces a human user to look for and manually delete stale files. Transactional file systems do not provide *new* crash-consistency guarantees for these applications; rather, they remove the burden of recovery and cleanup, simplifying the application and eliminating bugs [PCA+14].

2.2 File system indexes

File systems need to maintain indexes for metadata and data in order to search for the target data structures for the system calls to work on. The metadata indexes help look up metadata structures such as inodes and directory entries (dentries). The data indexes help look up file data blocks. This dissertation focuses on improving the metadata indexes. In this section, we introduce the implementation of metadata and data indexes in existing file systems, and discuss why the Virtual

File System (VFS) layer can become a performance and scalability bottleneck for metadata operations.

2.2.1 File system metadata indexes

File systems rely on multiple data structures for metadata maintenance and lookup. All file systems in the linux kernel are built on top of the VFS. The VFS layer defines the system call interfaces to the user space, and works as an abstract layer on top of the underlying file systems. It supports file table management, which tracks open files per kernel thread, and the files' read/write offsets. It also provides caching and concurrency control, enabling parallel accesses to the shared resources in a file system. VFS maintains DRAM caches for its internal data structures, including inodes, dentries, buffer heads and data pages. The VFS layer helps system calls look up file inodes with the following steps.

(1) The VFS pathname lookup starts by searching the VFS dentry cache (dcache) for the file dentry. It decomposes the file path into directories and file name components, and searches the dentries component by component, starting from the root directory if provided with an absolute path; or from the current directory '`.`' or the parent directory '`..`' if provided with a relative path. This step is called a path walk. If the target dentry is found after the path walk, and that dentry points to an inode, the dentry is called a positive dentry and the lookup procedure returns;

(2) If the dentry retrieved from the previous step keeps a NULL pointer to the associated inode, it is called a negative dentry. This leads to step (4);

(3) If the target dentry does not exist in the dentry cache, a new dentry is allocated for the file. The dentry at this point is a negative dentry;

(4) The VFS searches its inode cache to check if the target file inode is already cached. If it is, the inode is associated with the dentry for the convenience of later pathname lookup. The dentry turns into a positive dentry and gets returned. Otherwise, the inode is not cached and it goes to step (5);

(5) The VFS queries the underlying concrete file system to lookup the inode from storage devices. If the inode exists on disk, the file system loads the inode and caches it in the DRAM inode cache. The inode is then pointed by the dentry and the dentry becomes positive. Otherwise, if the file inode does not exist in the on-disk file system, the file system will try to allocate a new file inode. This leads to the next step;

(6) When creating the new file, the VFS layer first checks the system call flags. If the `O_CREAT` flag is not provided, the system call is returned with a `-ENOENT` error code. Otherwise, it requires the underlying file system to create the inode on disk. The new inode is then cached in the inode cache and pointed by the file's dentry. So the dentry becomes positive and can be utilized for future path lookups.

There are several performance and scalability bottlenecks caused by the metadata lookup and concurrency control mechanisms in the VFS. The VFS layer uses a dentry hash table and an inode hash table to index the two caches. A global

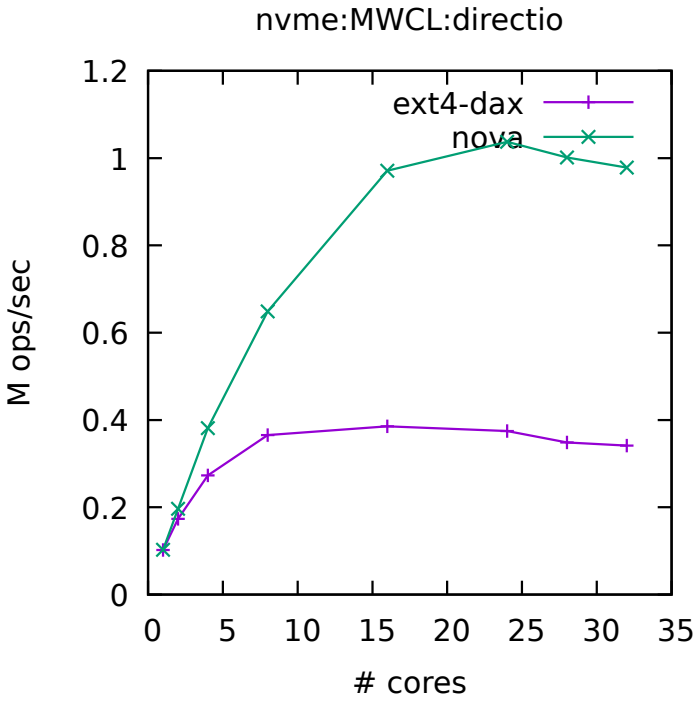

Figure 2.3: Ext4-dax and NOVA file creation scalability with FxMark MWCL workload. Multiple threads create files in their private directories.

inode hash lock is used to protect the inode hash table and it serializes each inode insertion and deletion. During a file creation, the file's directory entry needs to be inserted into its parent directory's dentry data page. This operation requires it to hold the reader-writer lock of the dentry data page. The lock further decreases the file system metadata scalability in related workloads.

We measured the scalability of metadata-related file system calls for both Ext4-dax and NOVA on a machine with 76GB DRAM, 2934 MT/s and an AMD Ryzen Threadripper 299WX processor, with 32 physical cores, no hyperthreading. File create, unlink and rename operations in a shared directory only scale up to 2

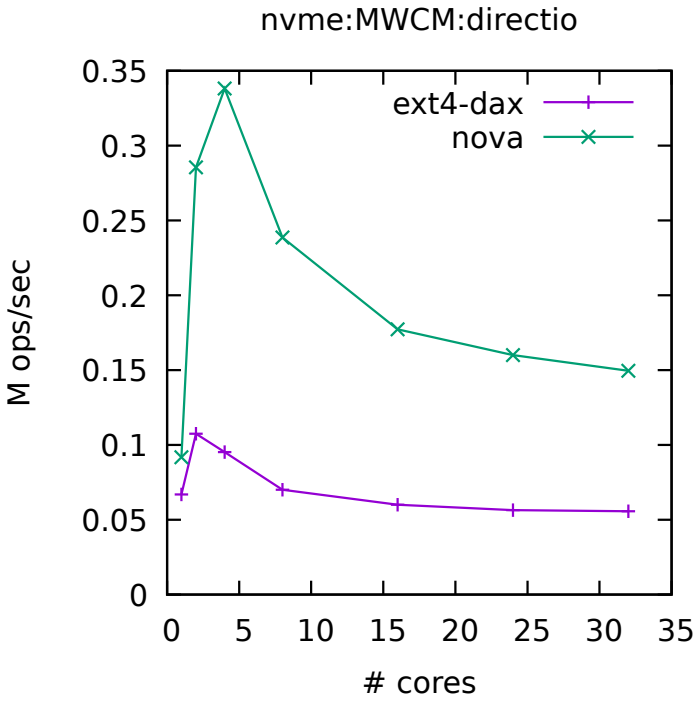

Figure 2.4: Ext4-dax and NOVA file creation scalability with FxMark MWCM workload. Threads create files in a shared directory.

cores. Operation throughput starts dropping with $\geq$2 cores.

We then port FxMark [MKMK16] to run tests on Ext4-dax and NOVA. FxMark's MWCL workload allows each thread to create files in a private directory. With MWCL, Ext4-dax scales up to 4 cores and NOVA scales up to 20 cores.

FxMark's MWCM workload creates files in a shared directory. With MWCM, Ext4-dax scales up to 2 cores and NOVA scales to 4 cores. Figure 2.3 shows the results for the MWCL benchmark while Figure 2.4 shows the results for MWCM.

2.2.2 File system data indexes

Data indexes help file systems look up data blocks for both directories and non-directory files. The page cache is a in-DRAM cache for data pages. File systems built on the page cache rely on the per-file red-black trees to index DRAM-cached data pages. DAX file systems like ext4-dax, xfs-dax and NOVA bypasses the page cache. They rely on the Direct Access (DAX) mechanism supported by some storage devices to perform reads and writes directly, without creating extra copies in DRAM.

A file system's on-disk data indexes depend on the file system specific implementation. For example, ext4 uses per-file extent trees to index on-disk data blocks. NOVA [XS16] is a log-structured file system with per-inode logs on disk. Since logs are not efficient data structures for indexing. NOVA maintains in-DRAM red-black trees for data page indexing.

2.3 Persistent memory

Persistent Memory is a non-volatile storage device. Its performance characteristics lie between DRAM and block devices. Compared to HDDs and NAND flash devices, PMEM provides orders of magnitude higher bandwidth and lower latency. It is slower than DRAM, but provides affordable large memory capacity [IYZ+19]. Persistent memory brings new opportunities and challenges to the file system design. PMEM devices enable byte-addressable memory accesses, which allows loads and stores with byte-level granularity and eliminates the need for

DRAM caches that are so helpful for traditional block devices.

While PMEM is high performance, it does have certain performance quirks. Its bandwidth is asymmetric with respect to access type. The read bandwidth for NVM can be nearly two times higher than its write bandwidth. We focus on optimizing read-heavy workloads and metadata-heavy workloads in which write sizes are small because we are primarily interested in multi-core scalability.

Intel Optane DC persistent memory module [Int] is a commercially available PMEM. It can be installed and configured into two modes, Memory Mode and App Direct mode. In Memory Mode, the Optane DC PMEM acts as the volatile main memory, and the DRAM works as an extra layer of the memory cache. In App Direct mode, the Optane DC PMEM acts as byte-addressable persistent memory, and can be viewed by applications and the operating system as a direct accessible storage module. In our case, we configure the Optane DC PMEM module in App Direct mode.

Research [WJX18, VNP+14] shows that the performance overhead in the VFS layer becomes more significant when moving from block-based devices to persistent memory. PMEM's decreased device latency reduces the IO overhead, making the CPU overhead a new bottleneck. As a result, our work tries to eliminate the VFS layer and substitute the metadata indexing with other more scalable data structures that rely on low-level memory ordering primitives.

Similar to DRAM, the store instructions in PMEM are ordered by the memory fences in x86 architecture. `CLFLUSH`, `CLFLUSHOPT` and `CLWB` instructions

flush a cache line to the persistent memory controller write pending queue (WPQ), where data is guaranteed to be power-fail safe (called the power-fail safe persistence domain). Compared to `CLFLUSH`, `CLFLUSHOPT` and `CLWB` are new optimized cache flushing instructions without serialization. The latter two instructions need to be serialized by a `SFENCE` instruction. A `CLWB` instruction invokes a cache line write-back. While both `CLFLUSHOPT` and `CLWB` followed by a `SFENCE` flush the cache line and clear its dirty bit, `CLWB` does not invalidates the cache line, making future accesses more likely to get a hit. Hence, a combination of `CLWB` and `SFENCE` achieves durability with low latency and is used in our proposed system.

2.4 Persistent hash tables

Our system design for file system metadata indexes is based on the P-CLHT [LMK+19] hash table. P-CLHT is a high performance persistent hash table with crash consistency guarantees. It is the persistent version of the Cache Line Hash Table (CLHT) [DGT15]. CLHT is a DRAM hash table that focuses on optimizing cache line operations. It allocates data buckets with the size of a cache line, so that most hash table operations lead to at most one cache line transfer, which is the granularity of the cache-coherence protocols, CLHT implements lockless searches and lock-protected updates. P-CLHT adds crash consistency to CLHT by inserting memory fences for ordering and cache line flushes to persist the store instructions. P-CLHT hash table maintains crash consistency with high performance by following the followed rules [LMK+19].

(1) Reads are non-blocking;

(2) Writes can be blocking or non-blocking;

(3) Updates to the hash table are made visible to other threads with a single hardware-atomic store;

(4) The atomic store is followed by a cache line flush and a memory fence, which flushes dirty data to PMEM in the same order as the writes to the CPU cache.

P-CLHT's write operation is synchronized by a per-bin lock. Its read operations are non-blocking, and are guaranteed to be consistent by snapshot checking. When a read operation finds a key in the hash table, it first creates a snapshot of the value before checking the key. If the key matches, it returns the snapshot value. The write operations also need to insert a `MFENCE` after writing a value and before inserting its key. This guarantees the consistency of the read operations. P-CLHT resizes its hash table when the number buckets exceeds a threshold. The resize operation creates a new hash table with an increased number of buckets, copies all the buckets to the new hash table and finally performs a atomic pointer swap to switch to the new table. Resize also holds the per-bin lock. It serializes with write operations, but can run in parallel with reads, since reads can still work on the old table. When a writer thread tries to write to a P-CLHT hash table which is under resize, the writer thread helps the resize procedure before acquiring the writer lock.

P-CLHT hash table serves as a good starting point for the NVMKVFS prototypical file system design because of its high throughput, lightweight implementation and simple crash consistency mechanism.

Chapter 3

TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions

TxFS is a transactional file system built from Linux's ext4 code base that uses ext4's journal to provide user-level transactions. TxFS's approach is practical; it tries to provide the greatest performance and scalability benefits to user-level transactions while minimizing implementation complexity by reusing the file system's journal. In ext4, the journal provides crash recovery, but TxFS extends the functionality to full ACID semantics.

Many modern applications must store structured data and perform rich, complex queries on that data. Many of these applications have turned to new abstractions such as embedded databases (*e.g.,* SQLite [SQL]) and key-value stores (*e.g.,* LevelDB [Goo]) to meet their data processing needs. The traditional file system API is not a good fit for these systems, which must resort to complex protocols that are inefficient and difficult to make correct (see §2.1.1). A transactional file-system interface makes these applications easier to write and makes them perform better on today's and tomorrow's hardware.

Another emerging feature of modern applications is their need to store data *across* different abstractions. For example, the Android Mail client stores email messages in SQLite, attachments as files, and the backup of mailbox information in

a key-value store [SBL+14]. For such applications, a single logical update from the user's point of view may require writes across different abstractions that all need to be performed atomically. Accomplishing this with user-level logs results in poor performance and added complexity (see §2.1.1).

The advantage to building TxFS on the file-system journal is that TxFS transactions obtain atomicity, consistency, and durability by placing each TxFS transaction entirely within a single file-system journal transaction, which is applied atomically to the file system. Using well-tested journal code to obtain ACD reduces the implementation complexity of TxFS, while limiting the maximum size of transactions to the size of the journal.

The main challenge of building TxFS is providing isolation. Isolation for TxFS transactions requires that in-progress TxFS transactions are not visible to other processes until the transaction commits. At a high level, TxFS achieves isolation by making private copies of all data that is read or written, and updating global data during commit. However, the naive implementation of this approach would be extremely inefficient: global data structures such as bitmaps would cause conflicts for every transaction, causing high abort rates and excessive transaction retries. TxFS makes concurrent transactions efficient by collecting logical updates to global structures, and applying the updates at commit time. TxFS includes a number of other optimizations such as eager conflict detection that are tailored to the current implementation of file-system data structures in ext4.

We find that the transactional framework allows us to easily implement a number of file-system optimizations. For example, one of the core techniques from

our earlier work, separating ordering from durability [CPADAD13], is easily accomplished in TxFS. Similarly, we find TxFS transactions allow us to identify and eliminate redundant application IO where temporary files or logs are used to atomically update a file: when the sequence is simply enclosed in a transaction (and without any other changes), TxFS atomically updates the file (maintaining functionality) while eliminating the IO to logs or temporary files (provided the temporary files and logs are deleted inside the transaction). As a result, TxFS improves performance while simultaneously providing better crash-consistency semantics: a crash does not leave ugly temporary files or logs that need to be cleaned up.

To demonstrate the power and ease of use of TxFS transactions, we modify SQLite, OpenLDAP and Git to incorporate TxFS transactions. We show that when using TxFS transactions, SQLite performance on the TPC-C benchmark improves by $1.6\times$ and a micro-benchmark which mimics Android Mail obtains $2.3\times$ better throughput. By porting OpenLDAP's LDIF backend to utilize TxFS transactions, OpenLDAP queries obtain crash consistency without losing performance. The TxFS version of LDIF provides the same guarantees as the Berkeley DB (BDB) backend, while introducing a performance boost up to $12.5\times$. Using TxFS transactions greatly simplifies Git's code while providing crash consistency without performance overhead. Thus, TxFS transactions provide crash consistency, reduce complexity, and increase performance.

We have made the following contributions.

- We present the design and implementation of TxFS, a transactional file system for modern applications built by leveraging the file-system journal (§3.1).

We have made TxFS publicly available at `https://github.com/ut-osa/txfs`.

- We show that existing file systems optimizations, such as separating ordering from durability, can be effectively implemented for TxFS transactions (§3.2).
- We show that real applications can be easily modified to use TxFS, resulting in better crash semantics and significantly increased performance (§3.3).

3.1 TxFS design and implementation

We now present the design and implementation of TxFS. TxFS avoids the pitfalls of earlier transactional file systems (§5): it has a simple API; provides complete ACID guarantees; does not depend on specific hardware; and takes advantage of the file-system journal and how the kernel is implemented to achieve a small implementation (≈5,200 LOC).

3.1.1 API

A simple API was one of the key goals of TxFS. Thus, TxFS provides developers with only three system calls: `fs_tx_begin()`, which begins a transaction; `fs_tx_commit()`, which ends a transaction and attempts to commit it; and `fs_tx_abort()`, which discards all file-system updates contained in the current transaction. On commit, all file-system updates in an application-level transaction are persisted in an atomic fashion – after a crash, users see all of the transaction updates, or none of them. This API significantly simplifies application code and provides clean crash semantics, since temporary files or partially written logs will

not need to be cleaned up after a crash.

`fs_tx_commit()` returns a value indicating whether the transaction was committed successfully, or if it failed, why it failed. A transaction can fail for three reasons: there was a conflict with another concurrent transaction, there is no journal space for the transaction, or the file system does not have enough resources for the transaction to complete (e.g., no space or inodes). Depending upon the error code, the application can choose to retry the transaction. Nested TxFS transactions are flattened into a single transaction, which succeed or fail as a unit. Flat nesting is a common choice in transactional systems [PHR+09, SGC+09].

A user can surround any sequence of file-system related system calls with `fs_tx_begin()` and `fs_tx_commit()` and the system will execute those system calls in a single transaction. This interface is easy for programmers to use and makes it simple to incrementally deploy file-system transactions into existing applications. In contrast, some transactional file systems (*e.g.,* Window's TxF [Rus05] and Valor [SGC+09]) have far more complex, difficult-to-use interfaces. TxF assigns a handle to each transaction, and requires users to explicitly call the transactional APIs with the handle. Valor exposes operations on the kernel log to user-level code.

TxFS isolates file-system updates only. The application is still responsible for synchronizing access to its own user-level data structures. A transactional file system is not intended to be an application's sole concurrency control mechanism; it only coordinates file-system updates which are difficult to coordinate without transactions.

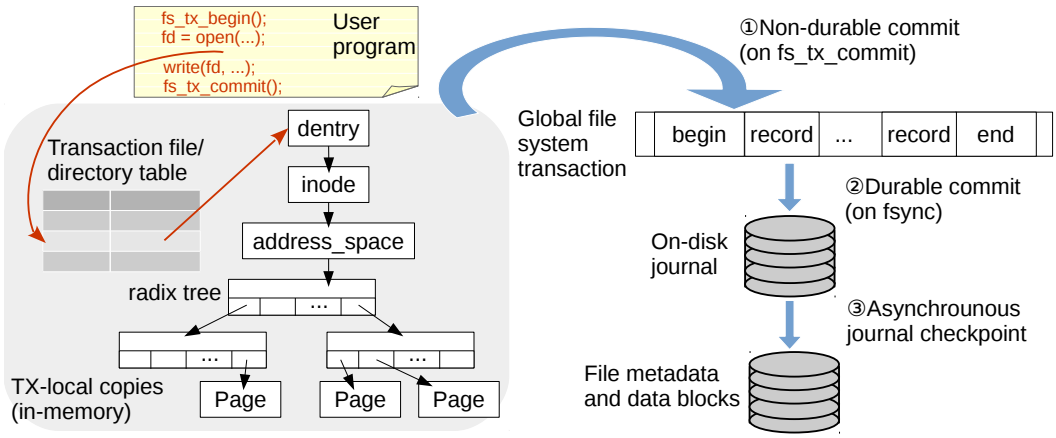

Figure 3.1: TxFS relies on ext4's own journal for atomic updates and maintains local copies of in-memory data structures, such as inodes, dentries, and pages to provide isolation guarantees. At commit time, the local operations are made global and durable.

3.1.2 Atomicity and durability

Most modern Linux file systems have an internal mechanism for atomically updating multiple blocks on storage such as journaling [Twe98] or copy-on-write [HLM94]. These mechanisms are crucial for maintaining file-system crash consistency, and thus have well-tested and mature implementations. TxFS takes advantage of these mechanisms to obtain three of the ACID properties: atomicity, consistency, and durability. This is the key insight which allows TxFS to have a small implementation.

TxFS builds upon the ext4 file system's journal. The journal provides the guarantee that each journal transaction is applied to the file system in an atomic fashion. We could have instead used a different mechanism such as copy-on-write which provides atomic updates in btrfs and F2FS. TxFS can be built upon any file

system with a mechanism for atomic updates.

For each TxFS transaction, TxFS maintains a private `jbd2` transaction, and at commit, merges the private transaction into the global `jbd2` transaction. While the global `jbd2` transaction contains only metadata by default, TxFS also adds data blocks to the transaction to ensure atomic updates. If, by chance, a block added to the private `jbd2` transaction is also being committed by a previous global `jbd2` transaction, TxFS creates a shadow block. Ext4 also creates a shadow block when a block is shared between a running and a committing transaction.

To reduce the amount of data written to the journal, TxFS employs selective data journaling [CPADAD13]. Selective data journaling only journals data blocks that were already allocated (*i.e.,* data blocks that are being updated), and avoids journaling newly allocated data blocks (because it can write them directly). Selective data journaling provides the same guarantees as full data journaling at a fraction of the cost.

TxFS ensures that an entire transaction can be merged into a *single* journal transaction; otherwise, an error is returned to the user. As long as a TxFS transaction is added to a single journal transaction, the journal will ensure it is applied to the file system atomically. After merging a user's transaction into the journal transaction, TxFS can persist the journal transaction, which ensures the durability of the TxFS transaction.

3.1.3 Isolation and conflict detection

Although the ext4 journal provides atomicity and durability, it does not provide isolation. Adding isolation for file-system data structures in the Linux kernel is challenging because a large number of functions all over the kernel modify file-system data structures without using a common interface. In TxFS, we tailor our approach to isolation for each data structure to simplify the implementation.

To provide isolation, TxFS has to ensure that all operations performed inside a transaction are not visible to other transactions or the rest of the system until commit time. TxFS achieves the isolation level of *repeatable reads* [GLPT76] using a combination of different techniques.

Split file-system functions. System calls such as `write()` and `open()` execute file-system functions which often result in allocation of file-system resources such as data blocks and inodes. TxFS splits such functions into two parts: one part which does file-system allocation, and one part which operates on in-memory structures. The part doing file-system allocation is moved to the commit point. The other part is executed as part of the system call, and the in-memory changes are kept private to the transaction.

Transaction-private copies. TxFS makes transaction-private copies of all kernel data structures modified during the transaction. File-system related system calls inside a transaction operate on these private copies, allowing transactions to read their own writes. In case of abort, these private copies are discarded; in case of commit, these private copies are carefully applied to the global state of the file system in an

atomic fashion. During a transaction, file-system operations are redirected to the local in-memory versions of the data structures. For example, dentries updated by the transaction are modified to point to a local inode which maintains a local radix tree which has locally modified pages.

Two phase commit. TxFS transactions are committed using a two-phase commit protocol. TxFS first obtains a lock on all relevant file-system data structures using a total order. The following order prevents deadlock: inode mutexes, page locks, inode buffer head locks, the global `inode_hash_lock`, the global `inode_sb_list_lock`, inode locks, and dentry locks. The Linux kernel orders the acquiring of inode mutexes based on the pointer addresses of their inodes; we adopt this locking discipline in TxFS. Similarly, page locks are acquired in order of page address. Acquiring the locks on buffers for directory data blocks and inode metadata is ordered by inode number.

After obtaining the locks, all allocation decisions are checked to see if they would succeed; for example, if the transaction creates inodes, TxFS checks if there are enough free inodes. Next, TxFS checks the journal to ensure there is enough space in the global `jbd2` transaction to allow the transaction to be merged. Finally, TxFS checks for conflicts with other transactions (as described below). If any of these checks fail, all locks are released, and the commit returns an error to the user. Otherwise, the in-memory data structures are updated, all file-system allocation is performed, and the private `jbd2` transaction is merged with the global `jbd2` transaction. At this point, the transaction is committed, locks are released and the changes are persisted to storage in a crash-consistent manner.

Conflict detection. Conflict detection is a key part of providing isolation. Since allocation structures such as bitmaps are not modified until commit time, they cannot be modified by multiple transactions at the same time, and do not give rise to conflicts; as a result, TxFS avoids false conflicts involving global allocation structures.

Conflict detection is challenging as file-system data structures are modified all over the Linux kernel without a standard interface. TxFS takes advantage of how file-system data structures are implemented to detect conflicts efficiently.

Conflict detection for pages. The `struct page` data structure holds the data for cached files. TxFS adds two fields to this structure: `write_flag` and `reader_count`. The `write_flag` indicates if there is another transaction that has written this page. The `reader_count` field indicates the number of other transaction that have read this page. Non-transactional threads will never see the in-flight un-committed data in transactions, and thus can always safely read data. TxFS does eager conflict detection for pages since there is a single interface to read and write pages that TxFS interposes. The following rules are followed on a page read or write:

1. When a transaction reads a page, it increments `reader_count` by one. If the page has the `write_flag` set, it must have been written by a currently executing transaction (which is now a conflict), so this transaction aborts.
2. If a transaction attempts to write a page that has either the `write_flag` set or `reader_count` greater than zero, it aborts. Otherwise, it sets the `write_flag`.
3. If a non-transactional thread attempts to write to a page with `reader_count` or `write_flag` set, it is put to sleep until the transaction commits or aborts.

4. When the transaction commits or aborts, `write_flag` is reset and `reader_count` is decremented.

Aborting transactions in this manner can lead to livelock, but we have not found it a problem with our benchmarks and the policy can be easily changed to resolve conflicts in favor of the oldest transaction (which does not livelock). TxFS favors transactional throughput, but for greater fairness between transactional and non-transactional threads, TxFS could allow a non-transactional thread to proceed by aborting all transactions conflicted by its operation [PHR+09].

Conflict detection for dentries and inodes. Apart from pages, TxFS must detect conflicts on two other data structures: dentries (directory entries) and inodes. Unfortunately, unlike pages, inodes and dentries do not have a standard interface and are modified throughout kernel code. Therefore, TxFS uses lazy conflict detection for inodes and dentries, detecting conflicts at commit time. At commit time, TxFS needs to detect if the global copy of the data structure has changed since it was copied into the local transaction. Doing a byte-by-byte comparison of each modified data structure would significantly increase commit latency; instead, TxFS takes advantage of the inode's `i_ctime` field that is changed whenever the inode is changed; TxFS simply has to check that the `i_ctime` has not changed for each inode that TxFS has read or written (writes are performed to a transaction-local copy of the inode). TxFS similarly adds a new `d_ctime` field to the dentry data structure to track its last modified time, and updates `d_ctime` whenever a dentry is changed. Creating different named entries within a directory does not create a conflict because the names are checked at commit time. By taking advantage of `i_ctime` and

`d_ctime`, TXFS is able to perform conflict detection for these structures without radically changing the Linux kernel.

Summary. Figure 3.1 shows how TXFS uses ext4's journal for atomically updating operations inside a transaction, and maintaining local state to provide isolation guarantees. File operations inside a TXFS transaction are redirected to the transaction's local copied data structures, hence they do not affect the file system's global state, while being observable by subsequent operations in the same transaction. Only after a TXFS transaction finishes its commit by calling `fs_tx_commit()` will its modifications be globally visible.

3.1.4 Implementation

We modified Linux version 3.18 and the ext4 file system. The implementation requires a total of 5,200 lines of code, with 1,300 in TXFS internal bookkeeping, 1,600 in the VFS layer, 900 in the journal (JBD2) layer, 1,200 for ext4 and 200 for memory management (all measurements with SLOCCount [Dav]). Except for the ext4 and jbd2 extensions, all other code could be reused to port TXFS to other file systems, such as ZFS, in the future.

3.1.5 Limitations

TXFS has two main limitations. First, the maximum size of a TXFS transaction is limited to one fourth the size of the journal (the maximum journal transaction size allowed by ext4). We note that the journal can be configured to be as large as required. Multi-gigabyte journals are common today. Second, although

parallel transactions can proceed with ACID guarantees, each transaction can only contain operations from a single process. Transactions spanning multiple processes are future work.

3.2 Accelerating program idioms with TxFS

We now explore a number of programming idioms where a transactional API can improve performance because transactions provide the file system a sequence of operations which can be optimized as a group (§2.1). Whole-transaction optimization can result in dramatic performance gains because the file system can eliminate temporary durable writes (such as the creation, use, and deletion of a log file). In some cases, we show that benefits previously obtained by new interfaces (such as `osync` [CPADAD13]) can be obtained easily with transactions.

3.2.1 Eliminating file creation

When an application creates a temporary file, syncs it, uses it, and then unlinks it (*e.g.,* logging shown in Figure 2.1b), enclosing the entire sequence in a transaction allows the file system to optimize out the file creation and all writes while maintaining crash consistency.

The create/unlink/sync workload spawns six threads (one per core) where each thread repeatedly creates a file, unlinks it, and syncs the parent directory. Table 3.1 shows that placing the operation within a transaction increases performance by $133\times$ because the transaction completely eliminates the workload's IO. While this test is an extreme case, we next look at using transactions to automatically

Workload	FS	TX
Create/unlink/sync	37.35s	0.28s (133×)
Logging	5.09s	4.23s (1.20×)
Ordering work	2.86 it/s	3.96 it/s (1.38×)

Table 3.1: Programming idioms sped up by TxFS transactions. Performance is measured in seconds (s), and iterations per second (it/s). Speedups for the transaction case are reported in parentheses.

convert a logging protocol into a more efficient update protocol.

3.2.2 Eliminating logging IO

Figure 2.1b shows the logging idiom used by modern applications to achieve crash consistency. Enclosing the entire protocol within a transaction allows the file system to transparently optimize this protocol into a more efficient direct modification. During a TxFS transaction, all sync-family calls are functional nops. Because the log file is created and deleted within the transaction, it does not need to be made persistent on transaction commit. Eliminating the persistence of the log file greatly reduces the amount of user data but also file system metadata (*e.g.,* block and inode bitmaps) that must be persisted.

Table 3.1 shows execution time for a microbenchmark that writes and syncs a log, and a version that encloses the entire protocol in a single TxFS transaction. Enclosing the logging protocol within a transaction increases performance by 20% and cuts the amount of IO performed in half because the log file is never persisted. Rewriting the code increases performance by 55% (3.28s, not shown in the table). In this case getting the most performance out of transactions requires rewriting

the code to eliminate work that transactions make redundant. But even without a programmer rewrite, just adding two lines of code to wrap a protocol in a transaction achieves 47% of the performance of doing a complete rewrite.

Optimizing SQLite logging with TxFS. Table 3.3 reports results for SQLite. "Rollback with TxFS" represents SQLite's default logging mode encased within a TxFS transaction. Just enclosing the logging activity with a transaction increases performance for updates by 14%. Modifying the code to eliminate the logging work that transactions make redundant increases the performance for updates to 31%, in part by reducing the number of system calls by $2.5\times$.

3.2.3 Separating ordering and durability

Table 3.1 shows throughput for a workload that creates three 10MB files and then updates 10MB of a separate 40MB file. The user would like to create the files first, then update the data file. This type of ordering constraint often occurs in systems like Git that create log files and other files that hold intermediate state.

The first version uses `fsync()` to order the operations, while the second uses transactions that allow the first three file create operations to execute in any order, but they are all serialized behind the final data update transaction (using flags to `fs_tx_begin()` and `fs_tx_commit()`). The transactional approach has 38% higher throughput because the ordering constraints are decoupled from the persistence constraints. The work that first distinguished ordering from persistence suggests adding different flavor sync system calls [CPADAD13], but TxFS can achieve the same result with transactions.

Experiment	TxFS benefit	Performance Improvement
Single-threaded SQLite	Faster IO path, Less sync	1.3×
TPC-C	Faster IO path, Less sync	1.6×
Android Mail	Cross abstraction	2.3×
OpenLDAP	Crash consistency, Scalability	12.5×
Git	Crash consistency	1.0×

Table 3.2: The table summarizes the micro- and macro-benchmarks used to evaluate TxFS, and the performance improvement (relative to ext4 in ordered journaling mode) obtained in each experiment.

3.3 Evaluation

We evaluate the performance and durability guarantees of TxFS on a variety of micro-benchmarks and real workloads. The micro-benchmarks help point out how TxFS achieves specific design goals while the larger benchmarks validate that transactions provide stronger crash semantics and improved performance to a variety of large applications with minimal porting effort.

Testbed. Our experimental testbed consists of a machine with a 4 core Intel Xeon E3-1220 CPU and 32 GB DDR3 RAM and a machine with a 6 core Intel Xeon E5-2620 CPU and 8 GB DDR3 RAM. All experiments are performed on Ubuntu 16.04 LTS (Linux kernel 3.18.22). The kernel is installed on a Samsung 850 (512 GB) SSD and all experiments are done on a Samsung 850 (250 GB) SSD. The experimental SSD is run at low utilization (around 20%) to prevent confounding factors from wear-leveling firmware.

Table 3.2 presents a summary of the different experiments used to evaluate TxFS and the performance improvement obtained in each experiment. In the Git experiment, TxFS provides strong crash-consistency guarantees without degrading performance. Note that if not explicitly mentioned, all our baselines run on ext4 with its default journaling mode, the ordered journaling mode.

3.3.1 Crash consistency

TxFS's ACID transactions should be recoverable after a system crash. In order to verify this crucial correctness property, we boot a virtual machine and run a script that creates many types of transactions in multiple threads with random amounts of contained work and conflict probabilities. We crash the VM at a random time and make sure the file-system journal is recoverable and that the file system passes all fsck checks. We have run over 100 random crashes and can recover the file system in all cases. An alternate way to test crash consistency would be to use a testing framework such as CrashMonkey [MC17, MMP+18].

3.3.2 Stress testing TxFS

We performed stress testing on TxFS to ensure its correctness in the face of conflicts and multi-threaded operations. Our stress tests had two main workloads. Our first workload was a micro-benchmark with six threads starting TxFS transactions and performing file-system operations picked at random across two files before committing. These threads generate a lot of conflicts, stressing TxFS conflict detection and isolation mechanisms. Our second workload uses the SQLite embed-

ded database, performing a number of database operations with multiple threads. We were able to run both workloads for over 24 hours on TxFS without a kernel crash or our unit tests failing, giving us a measure of confidence in the correctness and stability of the codebase.

3.3.3 SQLite

We modified SQLite to use TxFS transactions. SQLite parameters which control `fsync` frequency and checkpointing mode have a large impact on performance [PMC17]. We use `PRAGMA synchronous=NORMAL` (default) for all modes, and `PRAGMA wal_checkpoint(FULL)` for WAL mode to guarantee all ACID properties.

When SQLite uses TxFS transactions, crashes do not leave any residual files on storage. Currently, users often must remove these residual files by hand which is tedious and error-prone. TxFS transactions eliminate user-visible log files; user-level code sees only the before and after state of the database, not messy in-flight data.

Single-threaded SQLite. Table 3.3 shows that TxFS is the best performing option for SQLite updates. Data is the average of five trials with standard deviations below 2.2% of the mean. For the update workload, TxFS is 31% faster than the default. We report IO totals as part of our validation that TxFS correctly writes all data in a crash-consistent manner. Several choices for SQLite logging mode, including TxFS, result in similar levels of IO that resemble the no-journal lower bound. Write-ahead logging mode (WAL) writes more data for the insert workload,

	Performance (kOps/s)		IO (GB)		Sync/tx	
Journal mode	Insert	Update	Insert	Update	Insert	Update
Rollback (default)	53.9	28.0	1.9	3.9	4	10
Truncate	53.5 (0.99×)	28.9 (1.03×)	1.9	3.9	4	10
WAL	39.8 (0.74×)	34.6 (1.23×)	3.9	3.8	3	3
TxFS	51.4 (0.95×)	36.7 (1.31×)	1.9	3.8	1	1
Rollback with TxFS	52.1 (0.97×)	31.9 (1.14×)	1.9	3.8	1	1
No journal (**unsafe**)	54.9 (1.02×)	50.6 (1.81×)	1.9	1.9	1	1

Table 3.3: The table compares operations per second (larger is better) and total amount of IO for SQLite executing 1.5M 1KB operations grouping 10K operations in a transaction using different journaling modes (including TXFS). The database is pre-populated with 15M rows. All experiments use SQLite's synchronous mode (its default).

which harms its performance. Note that TXFS does not suffer WAL's performance shortfall on insert, and TXFS surpasses WAL's performance on update, making it a better alternative. Although the file-system journal shares similarity with a WAL log, TXFS does not generate redundant IO on insert because of its selective data journaling.

We run similar experiments with small updates (16 bytes) and find that there is little difference in performance between SQLite's different modes and TXFS. This shows that small transactions do not have significant overhead in TXFS.

TXFS's improves performance for the update workload due to several factors. TXFS reduces the number of data syncs from 10 (in Rollback and Truncate mode) or 3 (in WAL mode) to only 1, which leads to better batching and re-ordering of writes inside a single transaction. It performs half of its IO to the journal, which

is written sequentially. The remaining IO is done asynchronously via a periodic file-system checkpoint that writes the journaled blocks to in-place files. Since TxFS uses the file-system journal instead of an application-level journal for logging the transaction, it avoids the *double journaling problem* [SPZ14], where the journaling of the application-level log causes a significant slowdown. Even in realistic settings where performance is at a premium, transactions provide a simple, clean interface to get significantly increased file-system performance, while maintaining crash safety.

3.3.4 TPC-C

We run a version of the TPC-C benchmark [NPM+13] ported to use single-threaded SQLite[1]. TPC-C is a standard online transaction processing benchmark for an order-entry environment.

Table 3.4 shows that TxFS outperforms SQLite's default mode by $1.61\times$. The performance advantage comes from two sources. First, TxFS writes less data and batches its writes. TxFS writes much of its data sequentially to the file system journal on `fs_tx_commit()` and writes back the journal data asynchronously. SQLite's default mode must write data to the SQLite journal and to the database file on `fsync()`. Therefore, TxFS writes only once in the critical path (to the journal), while SQLite (as configured in Section 3.3.3) must write to the journal plus database in the critical path. Second, TxFS decreases the number of system calls, especially sync-family calls. Table 3.4 shows that TxFS reduces the number of sync-family calls per transaction by $3\times$. By reducing the sync-familly calls, TxFS

[1] `https://github.com/apavlo/py-tpcc/wiki/SQLite-Driver`

	Rollback (default)	Truncate	WAL	TxFS	No journal (**unsafe**)
Delivery	110.5	123.3	157.0	188.0	300.4
New Order	142.4	165.2	216.8	240.3	445.1
Order Status	1998.5	2067.3	2279.8	2489.9	3141.1
Payment	198.4	240.2	367.3	300.6	909.9
Stock level	575.0	602.3	765.4	684.1	1079.8
Total	173.0	(1.18×) 203.3	(1.62×) 280.0	(1.61×) 279.0	(3.47×) 600.1
Syscall/tx	208.0	207.95	138.26	100.35	146.9
Sync/tx	2.76	2.75	2.76	0.92	0.92
R KB/tx	18	17	13	13	7
W KB/tx	174	161	134	132	67
T KB/tx	192	(0.93×) 178	(0.77×) 148	(0.76×) 145	(0.39×) 74

Table 3.4: Rates (in transactions per second) for the TPC-C workload using different SQLite journaling modes. Each workload runs continuously for a fixed amount of time. Speedups (or, if less than 1.0, slowdowns) relative to Rollback mode are shown in parentheses. R MB/tx is the amount of read IO per transaction, W is written IO and T is total.

Journal mode	Throughput	IO(MB)
Rollback (default)	45.7	3269
Truncate	45.5 (0.99×)	3154
WAL	53.4 (1.17×)	3539
TxFS	105.7 (2.31×)	6797
TxFS Small Tx	60.9 (1.33×)	4052
No journal (**unsafe**)	61.9 (1.35×)	3995

Table 3.5: TxFS supports transactions across storage abstractions for a workload based on Android's mail application. Performance is measured in iterations per second.

can batch writes in a transaction, reducing the amount of writes by 31.7% compared to default mode.

The performance of TxFS and WAL is similar. When transactions contain writes, TxFS has better performance than WAL, but it has worse performance for read-only transactions: WAL is 28% faster than TxFS for read-only transactions. "Order status" and "Stock level" consist of three select queries and two select queries respectively, resulting in lower throughput for TxFS compared with WAL. However, "Delivery" consists of three select, three update, and one delete queries, so TxFS outperforms WAL by 20%.

3.3.5 Abstractions built on files

Modern file systems support storage of not only files but databases (*e.g.,* SQLite) and key-value stores (*e.g.,* LevelDB and RocksDB). These abstractions are built on the file system and generally are easier to set up and maintain (although lower-performing) than their dedicated counterparts.

TxFS supports transactions that span storage abstractions. Table 3.5 shows the throughput for a workload that models the core activity of Android mail, storing an image file and recording the path to that file in a SQLite database along with other metadata. The database is pre-populated with 100,000 1KB rows, image files are 1 MB. The workload creates the database record in one transaction, creates a uniquely named file where it stores the file data, syncs the data, and then updates the database record in a second transaction.

TxFS outperforms default SQLite by $2.31\times$ and the best alternative (WAL mode) by $1.98\times$. It is essential to TxFS's performance that both database transactions as well as the file system operation are all contained in a single transaction. When they are separate transactions (TxFS Small tx), performance is bounded by SQLite (i.e., it is close to no journaling). IO is not a bottleneck for this workload. The amount of IO performed is proportional to the amount of work done: TxFS has higher throughput, so it performs more IO.

3.3.6 OpenLDAP

OpenLDAP [Sym] is a widely-used, well-optimized implementation of the Lightweight Directory Access Protocol, used to provide authentication and auxiliary information about users. We used OpenLDAP version 2.4.44 and the LDIF backend which stores user records in plain text, one file per user, within a single directory. We modified approximately 155 LOC of the LDIF backend to use TxFS. We altered the LDIF backend to wrap transactions around the main backend calls (the `ldif_back_*` and `ldif_tool_entry_*` family of functions).

	Ext4-BDB		Ext4-LDIF (unsafe)		TxFS-LDIF	
	1-thread	4-thread	1-thread	4-thread	1-thread	4-thread
AddJob	597.9	304.8	2124.8 (3.6×)	3353.3 (11.0×)	2109.7 (3.5×)	3804.7 (12.5×)
ModifyJob	402.1	454.0	1063.2 (2.6×)	1402.5 (3.1×)	944.6 (2.4×)	2504.8 (5.5×)
DeleteJob	405.9	515.8	1972.8 (4.9×)	3861.1 (7.5×)	1701.1 (4.2×)	3893.1 (7.6×)

Table 3.6: Operations per second for OpenLDAP workloads of different operation types (larger is better). The base directory starts out with pre-populated entries. AddJob inserts 1,000 4KB entries. ModifyJob then modifies 1,000 disjoint entries. DeleteJob then deletes 1,000 entries.

Table 3.6 compares OpenLDAP using Berkeley DB (BDB) on Ext4 (Ext4-BDB), OpenLDAP using LDIF storage backend based on flat files on Ext4 (Ext4-LDIF), and a modified version of LDIF that uses TxFS transactions (TxFS-LDIF). We gather data using a modified version of the `lb` [Ham] benchmarking tool. After receiving a job request, Ext4-LDIF makes a temporary copy for the LDIF file storing the user information, modifies it, and replaces the original file with an atomic rename. This process is slow for large files. Also, it requires global locks to prevent race conditions from multiple slapd threads, which limits scalability. Both Ext4-BDB and TxFS-LDIF provide much stronger guarantees than Ext4-LDIF. A single job request is protected by an ACID transaction.

TxFS-LDIF always outperfoms Ext4-BDB with the same level of consistency guarantees. The throughput of TxFS-LDIF can be up to $4.2\times$ the throughput of Ext4-BDB in single-threaded workloads and $12.5\times$ in multi-threaded workloads. TxFS-LDIF provides comparable performance with Ext4-LDIF, while supporting

stronger consistency guarantees. Compared to Ext4-LDIF, TxFS-LDIF performance trails slightly in some of the workloads due to transaction overhead and the system call overhead to begin and end transactions. For multi-threaded benchmarks, TxFS transactions replace a global lock in the LDIF backend. The default implementation of OpenLDAP uses a single reader-writer lock for the whole backend directory, which limits concurrency by serializing writes. There are cases where concurrent writes are safe (*e.g.,* to two separate entries in two separate files), and TxFS allows these writes to occur concurrently, giving TxFS-LDIF higher multi-threaded write performance (5.5× BDB instead of 3.1× on ModifyJob).

3.3.7 Git

Git is a widely-used version control system. Git commands such as `git add` and `git commit` result in a large number of file-system operations. Git updates files by creating a temporary file, writing the desired data to it, and renaming it over the old file. To enable high performance, Git does not order its operations via `fsync()` [PCA+14], leaving it vulnerable to garbage files and outright data corruption on a system crash.

In our experiment, we run Git inside a virtual machine. We instrument the Git code to crash the virtual machine at vulnerable points (such as after the temp file rename, but before the file is persistent). The workload first initializes a Git repository, populates it with 20,000 empty files, then adds all files at once.

After a virtual machine restart, we find that the `.git/index` file has been truncated to zero bytes, resulting in a loss of the working tree. Running the Git re-

covery command `git fsck` simply reports a fatal error. Recovery is not possible unless the data has been backed up in another location. In contrast, when we change Git to use TxFS transactions, we find that crashes no longer produce such catastrophic errors. Furthermore, we do not find a significant difference in performance between the code that use TxFS transactions, and the code that does not. Thus, using TxFS transactions provides crash consistency for Git without any performance overhead.

Hash table	Key	Value
metadata	file path name	inode structure
data	<file_ino, block_id>	data page

Table 4.1: The table lists NVMKVFS's key and value types in the metadata and data hash tables.

Chapter 4

Fast metadata indexes for NVM file system

NVMKVFS is a user-space NVM file system prototype. Its goal is to demonstrate a set of key data structures we built for improving the metadata scalability for NVM file systems. It represents the whole file system as two global hash tables in persistent memory, one for metadata and the other for data indexing. Table 4.1 shows the keys and values in the two hash tables. The metadata hash table translates a full path name into an inode struct. The data page hash table translates the tuple of a file inode and a page number into a 4KB data page. NVMKVFS provides many POSIX APIs for file system system calls with crash consistency, including `getdents`, `stat`, `open` with file creation, `close` and `unlink`.

VFS for caching, path walk and concurrency control File systems in the Linux kernel rely on the VFS layer for caching and concurrency control. Most of the existing kernel file systems are designed for block devices such as hard drives and

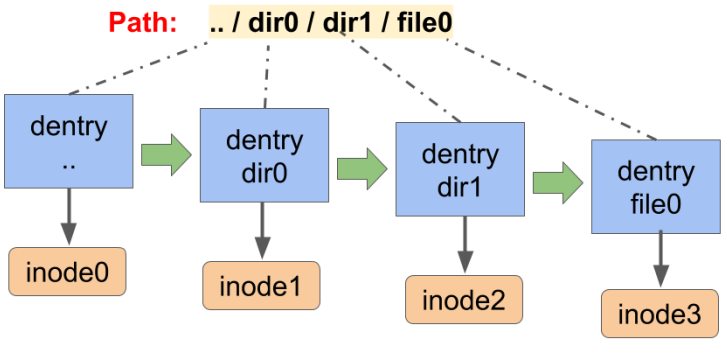

Figure 4.1: The VFS helps file systems to do path lookup with the help of the dcache. A path walk starts from the root directory for an absolute path or the current or parent directory for a relative path. It then walks the path component by component and searches for the dentries for the directories and files along the path, or loads the dentries if they are not in the dcache. The path walk ends when it reaches the target file.

NAND flash devices. Data on block devices needs to be retrieved and stored in block-sized granularity. In order to efficiently access file system metadata, which is generally much smaller than the block size, file systems pack metadata into blocks for storage and cache it in DRAM using the VFS. The inode cache helps look up the VFS inodes by indexing the inodes with their inode numbers. It is implemented as a hash table, in which each hash value is calculated from an inode number and the file system's physical device identifier. The hash table entries are pointers to the VFS inodes, which are contained in the file-system-specific DRAM-cached inodes. The VFS inodes can be referenced by dentry data structures, which are used in directory entry lookup during path resolution. VFS maintains a dentry cache in the DRAM to cache the translation from a path component to a dentry data structure. The dentry cache is also implemented as a hash table, while the hash value is a combination of

file or directory name with its parent directory's dentry virtual address. Figure 4.1 shows how the VFS and its dcache helps file systems do a path walk in order to find or create the dentry for a file. By resolving the path name component by component and traversing the dentry cache, a file system can find or create the dentry needed by a file that a system call needs to operate on. The VFS and its caches also provide concurrency control. Using multiple locks in the inodes, dentries and the per-file-system hash tables, processes can concurrently access a file system's in-memory and on-disk data structures in a safe way.

NVM challenges and opportunities NVM devices have byte-addressability, low access latency and high throughput, making the multiple caches in the file systems no longer necessary or efficient. It brings new challenges and opportunities to the file system design. DAX file systems, including ext4-dax, xfs-dax and NOVA, are designed for NVM devices. They bypass the page cache and instead use the direct access mechanism and load and store instructions to retrieve or access data in the NVM data pages. DAX file systems are built using the VFS layer and rely on its multiple caches for concurrency control. NVM's byte-addressability and low access latency, however, makes VFS metadata caching much less efficient [WJX18, VNP+14]. Furthermore, the concurrency mechanisms in the VFS restricts the parallelism for metadata-related operations. The global hash table locks, the per-superblock linked list for inodes and the multiple locks in the inode and dentry data structures all create serializing points in the file system operations.

As a result, we built NVMKVFS, a user-space file system protocol that by-

passes the VFS and provides its own mechanisms for metadata and data indexing and concurrency control. NVMKVFS targets improved scalability and performance of metadata-heavy workloads. On PMEM hardware, when running file creation workload in a shared directory, NVMKVFS gets a $7.45\times$ speedup over 28 cores compared to ext4-dax's $1.23\times$ speedup. On DRAM-emulated PMEM, the workload shows a $8.84\times$ 14-core speedup for NVMKVFS, compared to the $0.84\times$ speedup for ext4-dax and the $1.23\times$ speedup for NOVA.

4.1 NVMKVFS design and implementation

NVMKVFS maintains two persistent hash tables for metadata and data indexing. Both of the hash tables are variants of the p-CLHT [LMK+19] persistent hash table. NVMKVFS implements file system operations inside p-CLHT hash table operations. Based on the types of file system operations, we rewrite hash table `GET` method to get read-only operations and operations that only change an atomic value of an inode, and `PUT` method to get file system write operations. We follow the cache line write policy in the p-CLHT design, so that writes to the hash tables are the sizes of one or multiple cache lines. This resolves the cache coherence problem. We follow the locks and the orders of `MFENCE` and `CLWB` instructions in the p-CLHT hash table design to guarantee crash consistency for a single hash table read or write operation. For system calls that involve modifications on multiple key-value pairs in one or both hash tables, we use status bits in the inodes to help the recovery process after a crash. NVMKVFS does not cache inodes or directory entries in DRAM, so there is no caching layer for the file system data structures in

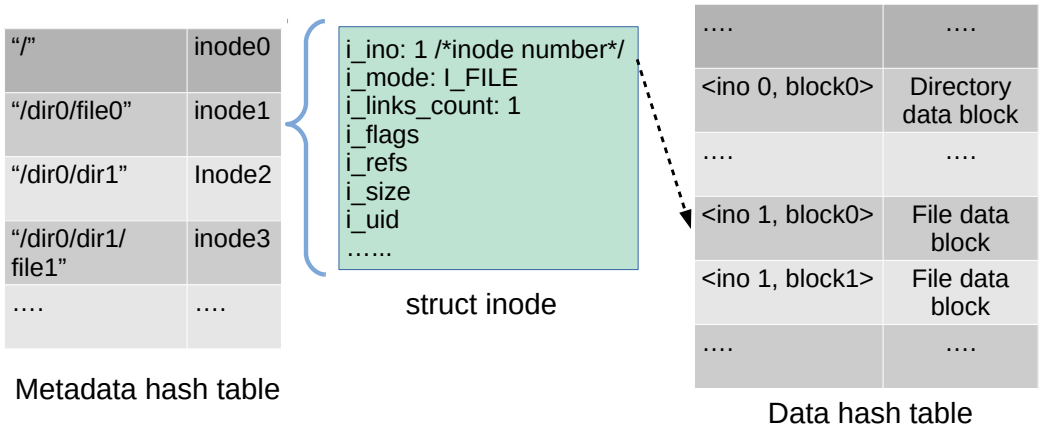

Figure 4.2: NVMKVFS represents the file system prototype with two hash tables, a metadata hash table which translates file paths to inode structures, and a data hash table which helps look up the file and directory data blocks. Both hash tables are variants of the p-CLHT persistent hash tables.

DRAM as in the VFS. To recover from a crash, NVMKVFS calls `fsck` on unclean reboots. Finally, metadata and data page hash table operations form the file system calls and provide POSIX-like APIs. NVMKVFS is currently only implementing metadata system calls. The implementation on the data-related system calls remains future work.

4.1.1 Data structures

Figure 4.2 shows the NVMKVFS storage layout and the data structures it uses to represent the prototype file system. The storage layout is mainly composed of two hash tables, one for metadata indexing and another for data indexing. Both of the hash tables are variants of the P-CLHT hash tables. File inodes are embedded as hash table values in the metadata table. Both file and directory data pages are stored as values in the data hash table.

Metadata index is a per file system hash table that translates a full path name into an inode structure. It is key to the performance of the metadata-heavy operations, such as `getdents`, `stat`, `open` with file creation, `unlink`, etc. The direct translation from a path name to an inode helps achieve fast metadata lookup but is not compliant with POSIX permission checking. In order to increase the metadata operation scalability, NVMKVFS's hash table implementation is based on the p-CLHT [LMK+19] persistent hash table, which implements lock-protected writes and lockless reads. We use djb2 hash function [djb] for string hashing. The update to a p-CLHT hash table usually occupies one cache line, so that it resolve the cache coherence problem. We follow these rules in the metadata hash table design, ensuring that write operations on file system metadata are protected by locks, while read-only operations are lockless. We also introduce a semi-read-only operation based on the read operations but updates a recoverable integer field with a hardware atomic operation. The crash consistency of metadata operations is guaranteed by following p-CLHT hash table recovery protocols. The buckets in the metadata hash table are all aligned to the cache line size during allocation.

Data index TxFS uses another hash table to index and allocate data pages. The data page hash table is also implemented based on the p-CLHT persistent hash table. A key in the data page hash table is a 64-bit integer formed by combining a file inode number (a 32-bit integer) and a page ID (a 32-bit integer). A value in the table is a fixed size data page. We use Jenkins' hash function [jen] for the 64-bit integer hashing. TxFS's current implementation uses 4kB data pages, while

the size of the data pages can be configured according to the file system's most frequent workloads. Since the data page hash table represents a 1:1 mapping from a key <file_ino, block_id> to a bucket containing a data page, it is also used as a block allocator. When allocating a new data block, the process queries the hash table to see if the block with a certain inode number and block ID exists. If the block does not exist in the hash table, the process allocates the page by creating a bucket with the key in the hash table. An offset in a file can then be translated into an offset in a data page in a hash table bucket. The bucket allocation in the hash table is protected by a per-bin readers-writer lock, which protects concurrent data block allocations with the same hash value. The buckets in the data hash table are all aligned to the cache line size during allocation.

Inode structure An inode is stored as the value field of a <pathname, struct inode> pair in the metadata indexing table. A NVMKVFS inode also records the followed properties to maintain correctness and crash consistency for metadata operations.

- Each file is allocated with an inode number; the inode number remains the same throughout the file's life span. It is unique to the file and serves partially as the key in the data index. NVMKVFS's current implementation allocates the inode number by atomically incrementing a global integer counter.
- A reference counter (refcount) records the number of processes opening a file, and are embedded in the inode structure in the metadata index.

- Each file entry's inode has an obsolete bit. When it is set to 1, it means it has already been removed by another process. This is to avoid the race condition between unlink and system calls involving metadata update when a file's refcount is not 0. The bit is also used in combination with a non-empty rename log for rename consistency.

- A creation bit is used to guarantee the crash consistency between the metadata and data indexes during a file creation. Although operations in each hash table are atomic due to the p-CLHT design, operations involving modifications on two hash tables cannot be guaranteed consistent without crash recovery. Before inserting a newly created inode into the metadata hash table, the inode's creation bit is set as 1. It is cleared after inserting the inode into the metadata table and writing the file name into its parent directory's data page. The inode insertion into the metadata table itself is atomic. So the inode creation bit indicates that a dentry re-insertion is necessary during a crash recovery.

- A per-file state lock enables parallel metadata operations. The lock is implemented with a hardware compare and swap operation. It has three possible states: free, update and resize. The resize state means that there is an ongoing hash table resize, and process updating the hash table needs to be either blocked or made to assist the resize procedure. The update state means that the hash table has an ongoing bucket update including bucket value updates and new bucket creation, and resize made by other processes needs to be blocked.

Data page The buckets in the data index contain data pages. Each data page contains a 64-bit integer metadata field. This integer is used as an offset counter during a directory entry insertion if the data page belongs to a directory.

DRAM file table TxFS maintains file tables in DRAM for file structure and descriptor allocation. During file system loading time, the file tables are allocated per each CPU in order to increase parallelism in file creation.

4.1.2 Directory layout

In NVMKVFS, directories and non-directory files share a single, global data page hash table. Directories have the same metadata and data format with non-directory files, except that they are indicated as directory types in their property fields and have different access permissions.

A directory data page contains dentries as a list of strings representing files in the directory. Each dentry string is a file name that occupies the size of a cache line, so that the cache coherence problem is resolved. Concurrent writes to a directory data page are protected by an atomic integer as the current write offset. Processes competing to write to the same directory data page will atomically increment the offset variable to get its own offset to write. If the offset reaches the size of a data page, a new directory data page is allocated and inserted into the data page index.

File metadata lookup in NVMKVFS is independent of the directory layout, since we rely on the global metadata index for path lookup. NVMKVFS implements

`getdents` by iterating over all the file dentries in the directory data pages, concatenating the file names with the directory path name to get the files' full path name, and looking up from the metadata index for the information of each individual file.

4.1.3 Metadata system calls

The metadata system calls are implemented as variants of p-CLHT `GET` and `PUT` operations. Read-only file system calls are implemented based on the hash table `GET` operation and are lockless. Updates are implemented based on the hash table `PUT` operation and holds a per-bin state lock, which protects the buckets with the same hash value. The per-bin locks protect concurrent update operations as well as concurrent update and hash table re-size operations.

open A non-creating `open()` first looks up the path name from the metadata store. If there is no such an entry with obsolete=0, -ENOENT is returned. Otherwise, it allocates a file structure and a descriptor in DRAM and holds a pointer to the metadata hash table entry. It then increments the inode's refcount to indicate that there is an active open.

close `close()` decrements and checks the file's refcount. If the refcount is greater than 1, it means some other process has opened the same file, so it returns. If the refcount is 1 and the inode's obsolete bit is on, it means some other process has already unlink-ed the file before. To finish up the incomplete `unlink`, it calls

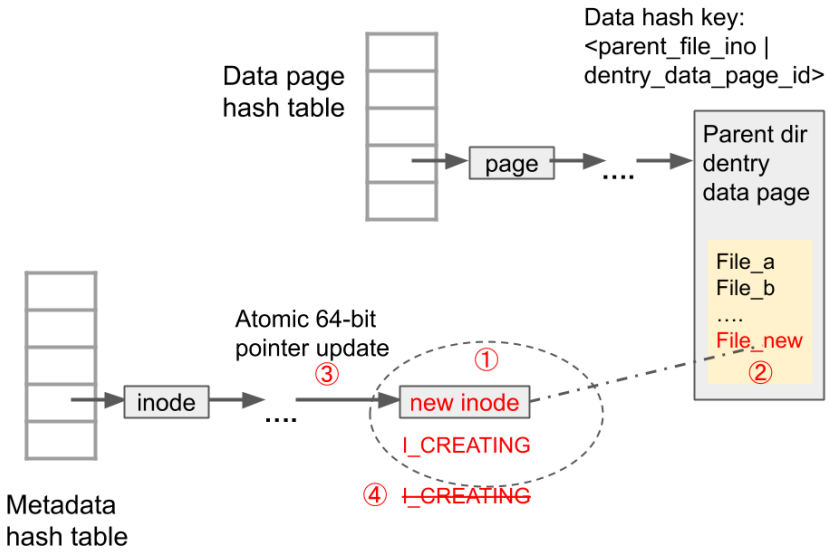

Figure 4.3: NVMKVFS creates a new file with the following steps: ① a hash table bucket containing the new inode is created with the I_CREATING flag on; ② the new file name is inserted into the parent directory's dentry data page; ③ the bucket containing the new inode is inserted into the metadata hash table with a single atomic pointer update; ④ the I_CREATING flag of the inode is cleared.

`unlink` to remove the file inode entry from the metadata index, Finally, it cleans up the file descriptor and frees the file struct.

access Performs the same operations as `open`, except for allocating file descriptors and incrementing the refcounts. It returns `F_OK` if the file exists and can be accessed.

create A open() system call with O_CREAT first looks up the inode structure from the metadata store.

(a) If there is no entry with the same path name, create and insert an entry;

(b) If there is an entry with the same path name and obsolete=0, open and return a file descriptor without creation;

(c) If there is an entry with the same path name and obsolete=1, keep on scanning the bucket until finding an entry with obsolete=0. Then, open and return a file handler without creation;

(d) If all entries in the bucket are with obsolete=1, create and insert a new entry.

Figure 4.3 shows how NVMKVFS creates and inserts a new file entry in the metadata hash table and inserts its dentry into its parent directory's dentry data page. NVMKVFS creates an inode by allocating a new metadata hash table bucket. NVMKVFS then looks up the file's parent directory data page in the data hash table and appends the new file's dentry at the end of the page. It then inserts the bucket containing the inode into the metadata hash table with a 64-bit pointer update, which is the unit of an atomic memory store for aligned data. An `I_CREATING` flag is used to maintain the crash consistency between the entries in the two hash tables. The details on the NVMKVFS crash consistency will be discussed in Section 4.1.4.

unlink It first looks up the inode structure from the metadata store. The metadata table supports multiple inodes with the same path name as long as only ¡=1 entry is with a non-zero obsolete bit.

(a) If a file entry is looked up with obsolete=0 and refcount=0, unlink() first removes the file path from the parent directory's dentry data page. Then the

file entry can be removed from the metadata store. A file unlink bit is set during the unlink operation so that if there is a crash during file creation, the recovery process can help finish the unlink.

(b) If a file entry is looked up with obsolete=0, and the refcount¿0, it sets the obsolete=1 so that the last process that closes the file will do the clean up;

(c) If a file entry is looked up with obsolete=1, the process continues to search until finding a entry with obsolete=0, and does step (a) if refcount=0 or step (b) if refcount¿0;

(d) If all file entries with the path name have obsolete=1, do nothing.

stat The `stat` system call looks up an inode in the metadata hash table according to its path name, and then fills up the stat structure to be returned.

getdents A directory file's data pages contain lists of file names for the files inside that directory. When `getdents` traverses a directory, the process first looks up the parent directory's inode from the metadata store. Then it transverses the directory by iterating over dentries in all its data pages, and looks up the file entries from the metadata store.

4.1.4 Crash consistency in NVMKVFS

Section 2.4 lists several rules for the p-CLHT hash table to maintain crash consistency. By following those rules, a p-CLHT hash table allows its entries to

change from an initial state to a final state with a single atomic step, so that a crash at any point leaves the hash table consistent. This eliminates the needs for crash recovery in the original p-CLHT design.

NVMKVFS follows the p-CLHT rules for crash consistency by constructing file system operations based on the p-CLHT hash table operations. The basic file system operations in the NVMKVFS hash tables can be placed into three categories: read-only, update and hybrid. Read-only operations are built from p-CLHT hash table `GET` operation and are lockless; update operations are built from p-CLHT hash table `PUT` operations; hybrid operations only modify a single 64-bit field, which can be achieved by updating the field with a single atomic store and building the rest of this file operation. For file system operations requiring crash consistency between multiple metadata entries, or even between metadata and data entries, NVMKVFS relies on status bits to assist the crash recovery after a power failure or crash.

Similar to ext4's default crash consistent mode, write-back mode, NVMKVFS guarantees only metadata consistency. It has no guarantee on data consistency. NVMKVFS follows the p-CLHT rules for crash consistency to implement file system calls. Updates to the metadata and data hash tables are all implemented with single hardware-atomic stores. For example, when NVMKVFS allocates or deallocates an inode, the updates to the metadata hash tables are implemented with an atomic pointer swap. This is the same for data page allocation and deallocation. For inode field updates that involve single atomic stores, NVMKVFS inserts a memory fence followed by a cache line flush to guarantee the write ordering. Since data pages in a directory file also contain metadata information such as dentries, it also

requires consistency between inodes in the metadata table and the dentry information stored in the directory data pages. In this case, NVMKVFS utilizes status bits to assist the recovery process if there is a crash during the file creation. For example, when creating a file, NVMKVFS first allocates an inode with `I_CREATING` bit on. After initializing all the inode fields, NVMKVFS creates and inserts the file's data pages into the data hash table before inserting the inode into the metadata hash table. The insertion of an inode into the metadata table is a single atomic store that writes a 64-bit address of the inode. Finally, NVMKVFS removes the `I_CREATING` bit from the inode.

NVMKVFS relies on `fsck` for crash recovery. `fsck` is invoked on recovery after an unclean `umount` or crash. It scans all the entries in the metadata hash table, and sets the entry's reference counter to 0. For each inode, it recovers inconsistent metadata updates.

- If a file inode has its creation bit on, it means a crash happened during a file creation and after the newly allocated inode is inserted into the metadata index. In this case, `fsck` checks and re-inserts the file's name into its parent directory's data page;
- If a file entry has obsolete=1, `fsck` checks whether it has been removed from the parent directory's dentry data pages. If not, it removes the file dentry;
- If a file inode has obsolete=1 and has a non-empty rename log, `fsck` finishes the rename and removes the obsolete entry;
- For each inode structure, clear its refcount and stale lock.

4.1.5 Discussions

4.1.5.1 Block allocators

Similar to P-CLHT and other persistent indexes designed for PMEM, NVMKVFS currently uses the `libvmmalloc` allocator from the Persistent Memory Development Kit (PMDK) [pmd]. The library is preloaded using `LD_PRELOAD`, and translates `malloc`, `memalign` and other memory allocation functions into the correspondent PMEM version in the `libvmmalloc` library. It allocates memory from a preallocated file on the PMEM device as a memory pool. The mapping from a PMEM virtual address to a file offset can change after the `libvmmalloc` library reload. This problem can be solved by either porting NVMKVFS from virtual address-based to offset-based programming, by substituting PMEM virtual addresses into the offsets in the memory pool file on the PMEM device. `libpmemobj` in PMDK is also a good substitute, since it allocates memory objects with a designated ID. We can also consider using a PMEM allocator such as Makalu [BCB16]. However, there is currently no standard and stable PMEM allocator implementation.

4.1.5.2 Trade-off between performance and POSIX-compliance

Compared to hash tables, tree data structures used in the traditional file systems can decrease the throughput and scalability of indexing. In order to look up a leaf node, search operations need to iterate over the tree nodes from the root to the leaf, leading to higher computational complexity. In each iterating step, read and write operations may acquire locks to protect concurrent access to the tree, which

Machine	PMEM	CPU	DRAM
PMEM	931GB Intel Optane DC, 4 nodes	Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz, 4 sockets × 28 cores, no hyperthreading	2.1TB
Emulated PMEM	50G emulated PMEM	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 1 socket × 14 cores, 2 hyperthreads/core	64GB DDR4

Table 4.2: The table lists the configurations for the machines used in our testbed.

can introduce scalability bottlenecks. Regular hash tables, on the other hand, have O(1) amortized time complexity for searching. By constructing hash table keys as file path names, NVMKVFS gets O(1) amortized time complexity for path lookup. However, by using full path names as keys, NVMKVFS loses its compliance with the POSIX permission checking. In the POSIX standard, permission checking is done component-by-component on each directory or file along a path walk. The tree representation for paths by nature supports such a permission checking with search operations. This is due to the tree-analogous nature of the Unix-style file paths. To summarize, there exists a trade-off between performance and POSIX-compliance when choosing between trees and hash tables for the file system path lookup.

4.2 Evaluation

We evaluate the throughput and scalability of NVMKVFS on metadata-heavy micro-benchmarks. The micro-benchmarks help us understand how NVMKVFS performs in our target workloads as well as how it compares to other VFS-based file systems. NVMKVFS achieves better scalability by removing the caching layers like

VFS, so that the synchronization points and scalability bottlenecks are eliminated.

NVMKVFS is currently a file system prototype that focuses on metadata operations. The current implementation supports system calls including `access`, `stat`, `creat`, `open`, `close`, `unlink` and `getdents`. The system calls are supported by the POSIX system call interfaces. The following sections evaluate the throughput and scalability of NVMKVFS with file creation under different conditions, file open-close and file stat workloads. The result shows that NVMKVFS outperforms ext4-dax and NOVA in most of the cases. NVMKVFS has better scalability in file creation and file open-close workloads compared to the other two file systems.

Testbed. Our experimental testbed consists of two machines, one with PMEM hardware and the other with root access for DRAM-emulation tests. We do not have root access to the machine with PMEM, which is necessary for the NOVA [XS16] kernel installation. So we only run NVMKVFS and NOVA comparison on this machine, and run NOVA comparison on the latter machine. Table 4.2 shows the machine configuration of the two.

4.2.1 Evaluation on PMEM hardware

This section describes the experiments run on the machine with Intel Optane DC. As shown in Table 4.2, the machine has 4 CPU sockets, each with 28 cores, and is installed with four Intel Optane DC persistent memory nodes. In order to avoid the impact of the NUMA effect, we run experiments with only 28 cores on the same socket, and use a single PMEM node that is local to that CPU socket. In

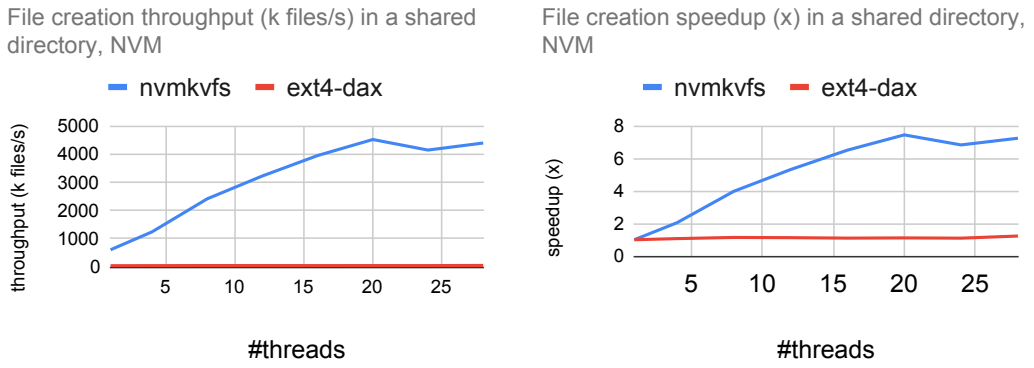

Figure 4.4: Throughput and speedup comparison between NVMKVFS and ext4-dax on the workload of file creation in a shared directory. Measured on Optane DC persistent memory.

this section, we only compare NVMKVFS with ext4-dax, since we do not have sudo access to this machine and cannot install the NOVA kernel.

4.2.1.1 File creation

Shared directory Figure 4.4 shows the performance comparison between NVMKVFS and ext4-dax with a file creation workload. The files are created in a single shared directory. NVMKVFS scales up to 20 cores, with a $7.45\times$ throughput increase, while ext4-dax scales poorly, with a $1.23\times$ speedup over 28 CPU cores. This is because in ext4-dax, the file creation is serialized by both locks in the VFS and locks specific to ext4. The locks in the VFS layer include the per-superblock inode hash lock and the parent directory's read-write semaphore. The ext4-related locks include per-block-group spinlocks for inode allocation and JBD2 journal-related locks. In the VFS, the inode hash lock is a per-superblock spinlock that protects the file system inode hash table. The parent directory's read-write semaphore is used to protect concur-

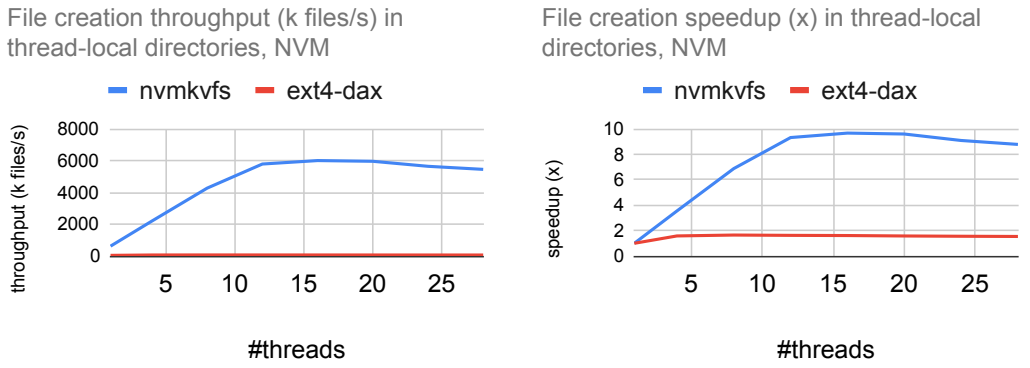

Figure 4.5: Throughput and speedup comparison between NVMKVFS and ext4-dax on the workload of file creation in thread-local directories. Measured on Optane DC persistent memory.

rent writes to the directory data page. These two locks are all parts of the VFS layer. Hence, all file systems built with the VFS should have the same performance bottleneck. NVMKVFS, on the other hand, relies on two persistent hash tables. It minimizes the sharing between sibling files. Threads performing file creation in the same directory are only serialized by a single atomic operation that increments the offset of the parent directory's dentry data page.

Thread-local directories Figure 4.5 shows the performance comparison between NVMKVFS and ext4-dax with the file creation workload, where each thread tries to create files in its thread-local directories. NVMKVFS scales up to 16 cores, with a $9.68\times$ speedup and 768 MB/s write throughput, while ext4-dax scales up to 6 cores, with a $1.65\times$ speedup. Though there is no contention on the parent directory's read-write semaphore, ext4-dax is still bottlenecked by the other locks, including the per-superblock inode hash lock in the VFS, and ext4-specific locks such as per-block-

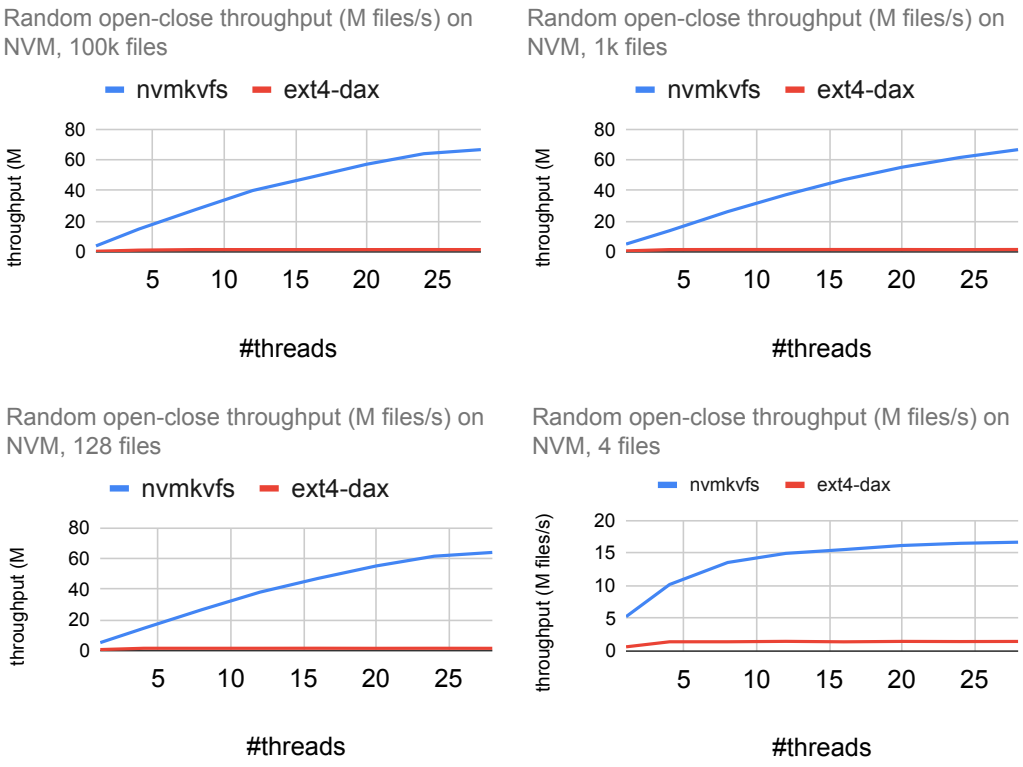

Figure 4.6: NVMKVFS and ext4-dax random file open-close throughput in a shared directory, with variant size of the file pool for shared accesses. Measured on Optane DC persistent memory.

group spinlocks for inode allocation and JBD2 journal-related locks. We are still investigating why NVMKVFS's throughput is saturated after 16 cores. This could be caused by the PMEM bandwidth, since the maximum write bandwidth of the Optane DC is 2.3 GB/s [IYZ+19]. We also tried tuning the hash table configuration, and were able to get better scalability with decreased throughput.

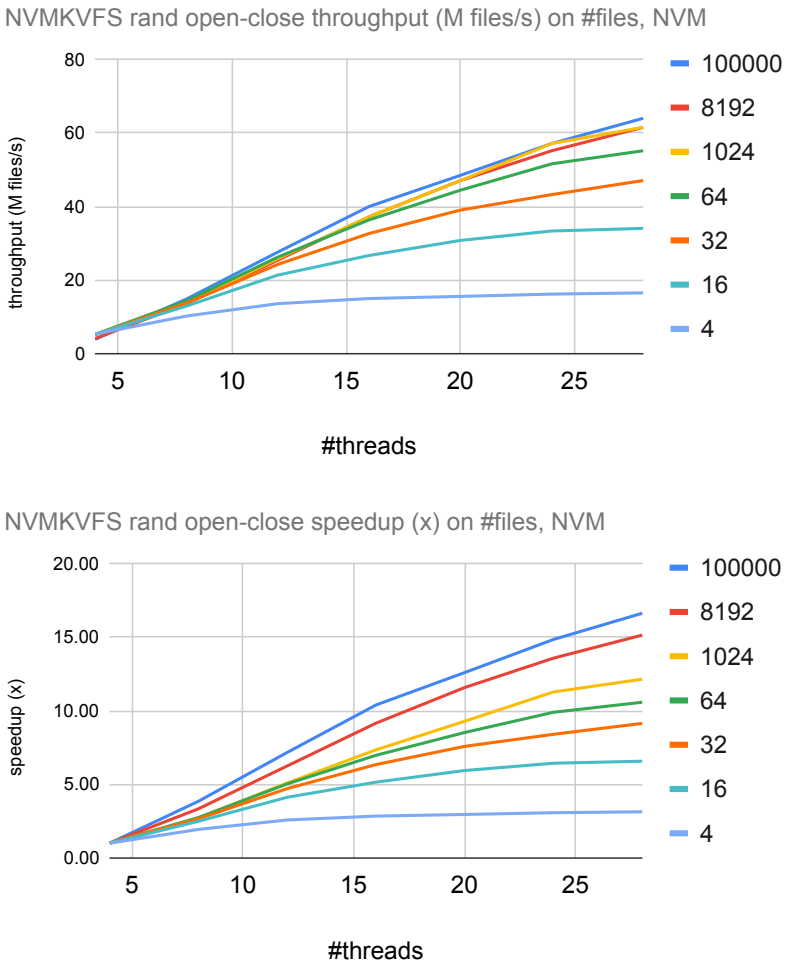

Figure 4.7: NVMKVFS random file open-close scalability on variant numbers of shared access files. Measured on Optane DC persistent memory.

4.2.1.2 Random file open and close

Figure 4.6 compares the performance between NVMKVFS and ext4-dax with the random file open and close workload. The workload starts by initializing a directory with a designated number of files. Then, multiple pthreads are allocated and

compete to open files that are randomly chosen from the directory. In NVMKVFS, file open is an operation implemented based on the P-CLHT hash table `GET` operation, which is a read-only operation with no locks being held, NVMKVFS's `open` operation also needs to increment an embedded reference counter in the NVMKVFS inodes in order to avoid the `unlink` operation deleting inodes that are currently held open by other threads. The reference counter is incremented with x86 hardware atomic instructions. As a result, this creates serialization on the atomic counters in the inodes, especially when multiple threads are competing to open the same file.

As illustrated in Figure 4.6, with different sizes of the shared directory, ext-dax scales up to 4 8 cores, while NVMKVFS scales to the maximum number of cores used in the tests. For example, with 128 files being shared, NVMKVFS gets a $11.36x$ speedup over 28 cores, while ext4-dax gets a $2.26\times$ speedup. NVMKVFS open-close scalability deprecates with decreased number of shared files. When using a pool of 100k, 1k, 128 and 4 files, NVMKVFS gets 28-core speedups of $17.33\times$, $13.21\times$, $12.36\times$ and $3.17\times$ respectively, while ext4-dax gets $3.94\times$, $2.47\times$, $2.26\times$ and $2.29\times$.

Figure 4.7 illustrates how NVMKVFS scalability changes with different sizes of the shared file pool. The single-thread open-close throughput decreases with increased number of shared files. In fact, the more files are initialized in the shared file pool, the higher the hash table's load factor. This increases the hash table bucket search time in read-heavy workloads. On the other hand, fewer files in the shared pool leads to higher probability of multiple threads competing on the same file, trying to increment the same atomic reference counter. This decreases the scalabil-

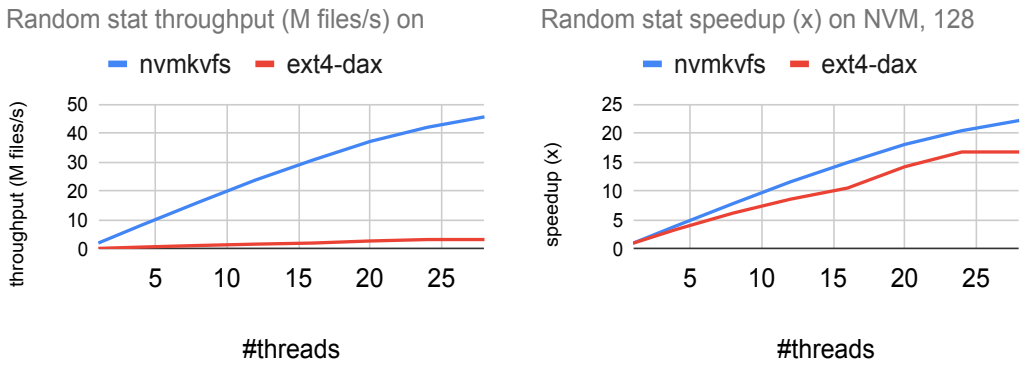

Figure 4.8: Throughput and speedup comparison between NVMKVFS and ext4-dax on the workload of retrieving status from files randomly picked in a pool of 128 files. Measured on Optane DC persistent memory.

ity. With 100k files being shared, NVMKVFS gets a $17.33\times$ speedup over 28 cores, while with 4 shared files, it gets a $3.17\times$ speedup.

4.2.1.3 Random file status retrieval

Figure 4.8 shows the performance comparison between NVMKVFS and ext4-dax with random `stat` operations. This is a workload on which NVMKVFS and ext4-dax both perform with high scalability, since it is a read-only workload. With 128 shared access files, NVMKVFS gets a $22.23\times$ speedup over 28 cores, while ext4-dax gets a $16.79\times$ speedup.

4.2.2 Evaluation on emulated PMEM

This section describes the experiments run on the machine with emulated PMEM. The machine does not have PMEM hardware. But the root access to the machine allows us to install and compare with the NOVA kernel file system. As

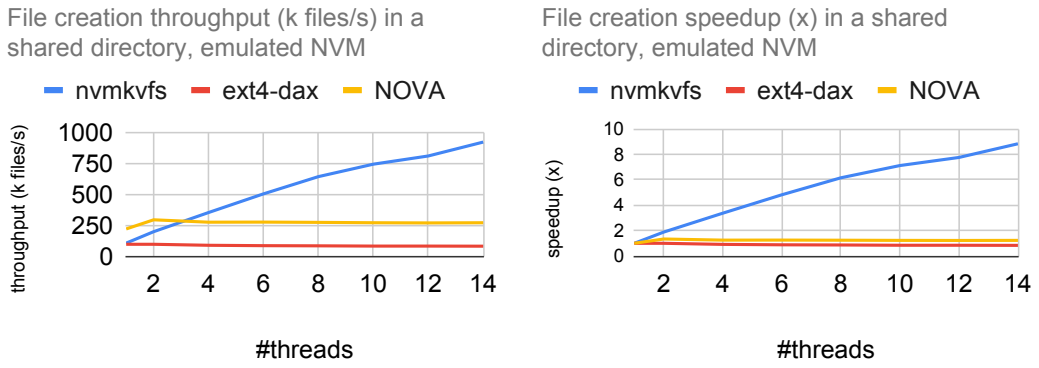

Figure 4.9: Throughput and speedup comparison between NVMKVFS, ext4-dax and NOVA on the workload of file creation in a shared directory. Measured on DRAM-emulated PMEM.

shown in Table 4.2, the machine has one CPU socket with 14 cores; each runs with two hyperthreads. We run experiments with the CPU ID-s on different physical cores to avoid the performance degradation with the hyperthreads. In this section, we compare NVMKVFS with ext4-dax and NOVA. The results do not show how the three file systems interact with the storage hardware; instead, it shows their CPU usage.

4.2.2.1 File creation

Shared directory Figure 4.9 shows the performance comparison between NVMKVFS, ext4-dax and NOVA with the file creation workload in a shared directory. NVMKVFS scales almost linearly to 14 cores, with a $8.84\times$ throughput increase, while ext4-dax does not scale, and NOVA scales up to 2 cores with a maximum of $1.34\times$ speedup. This is because both ext4-dax and NOVA are kernel file systems built under the VFS layer, with which file creation is serialized by the per-superblock inode hash

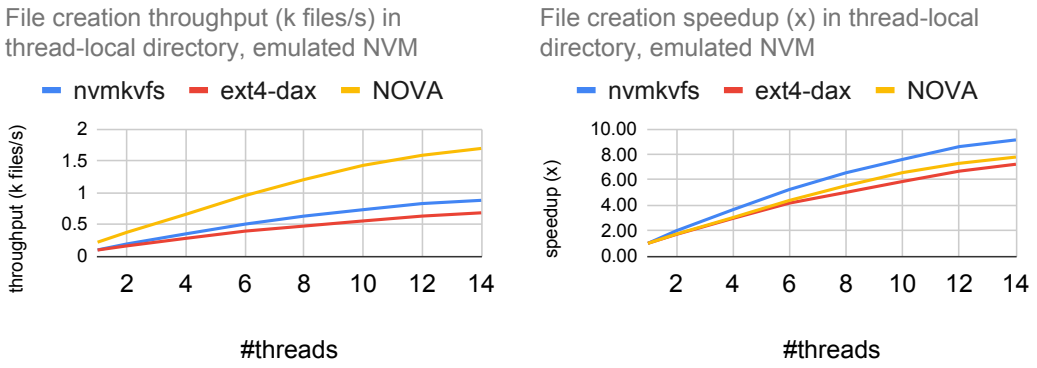

Figure 4.10: Throughput and speedup comparison between NVMKVFS, ext4-dax and NOVA on the workload of file creation in thread-local directories. directories. Measured on DRAM-emulated PMEM.

lock and the parent directory's read-write semaphore. All the file systems built with the VFS suffer from this bottleneck. NOVA's scalability is better than ext4-dax because of its per-inode metadata logging for crash consistency, per-CPU journal for across-inode logging and per-CPU inode table for inode allocation. These designs help NOVA avoid lock contention in journaling and inode allocation. NVMKVFS, on the other hand, relies on two persistent hash tables and avoids all the above-mentioned lock contention. In NVMKVFS, file creation in a shared directory is only serialized by the atomic increment operation on the write offset in the parent directory's dentry data pages.

Thread-local directories Figure 4.10 shows the performance comparison between NVMKVFS, ext4-dax and NOVA with the file creation workload in thread-local directories. All three file systems scale almost linearly. NVMKVFS, ext4-dax and NOVA respectively get a 14-core speedup of $9.15\times$, $7.23\times$ and $7.80\times$. Com-

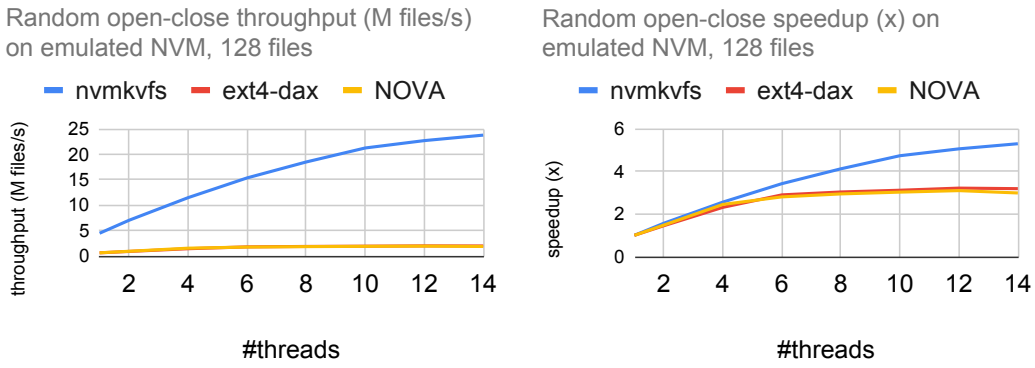

Figure 4.11: Throughput and speedup comparison between NVMKVFS, ext4-dax and NOVA on the workload of opening and closing files randomly picked in a pool of 128 files. Measured on DRAM-emulated PMEM.

pared to NVMKVFS and NOVA, ext4-dax is still bottlenecked by the ext4-specific locks such as per-block-group spinlocks for inode allocation and JBD2 journal-related locks. The scalability degradation in ext4-dax is not as obvious as in Section 4.2.1, because the experiment is run on the DRAM-emulated PMEM, which shortens the scopes of the critical sessions of the allocation and journal locks.

4.2.2.2 Random file open and close

Figure 4.11 compares the performance between NVMKVFS, ext4-dax and NOVA with the random file open and close workload. Both ext-dax and NOVA scales up to 8 cores, while NVMKVFS achieves a close-to-linear scalability over the 14 cores. This is because in NVMKVFS, the only serialization point in the workload happens when threads compete to increment the reference counter for the same file, while ext4-dax and NOVA are still bottlenecked by the VFS locks for shared resources. The 14-core speedups for NVMKVFS, ext4-dax and NOVA are $5.31\times$,

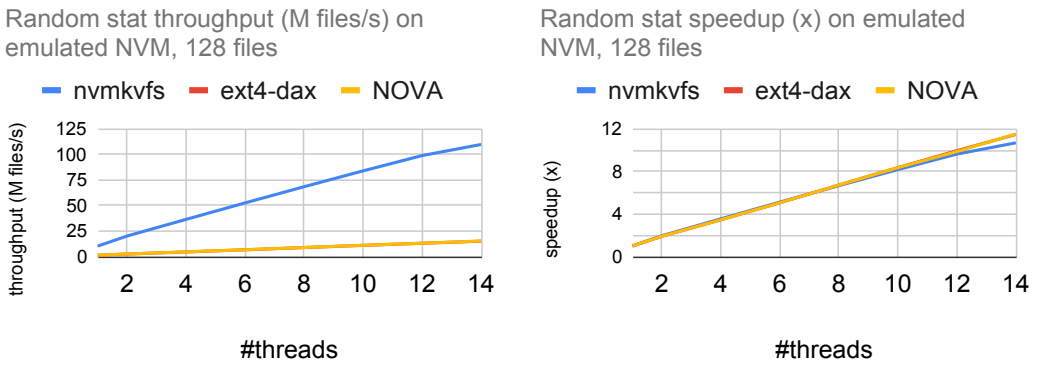

Figure 4.12: Throughput and speedup comparison between NVMKVFS, ext4-dax and NOVA on the workload of retrieving status from files randomly picked in a pool of 128 files. Measured on DRAM-emulated PMEM.

$3.20\times$ and $2.99\times$ respectively.

4.2.2.3 Random file status retrieval

Figure 4.12 shows the performance comparison between NVMKVFS, ext4-dax and NOVA with random `stat` operations. The three file systems get an almost linear and equally high scalability, since it is a read-only workload. With 128 shared access files, NVMKVFS, ext4-dax and NOVA get a $10.70\times$, a $11.49\times$ and a $11.53\times$ speedup over 14 cores.

Chapter 5

Related work

5.1 File system transactions

There have been a number of efforts over the years to provide systems support for file-system transactions. Each of these systems failed to gain adoption due to one of the following reasons: they had severe restrictions on what could be placed inside a transaction, they were complicated to use, they added complexity to the kernel, or they caused significant performance degradation. Learning from prior systems, TxFS avoids all of these mistakes. Table 5.1 summarizes related work and demonstrates that TxFS is unique among transactional file systems.

Building file systems on top of user-space databases. One way to provide transactional updates for applications is to build a file system over a user-space transactional database. OdeFS [Geh94], Inversion [Ols93], and DBFS [MTV02] use a database (such as Berkeley DB [OBS99]) to provide ACID transactions to applications via NFS. Amino [Wri07] tracks all user updates via `ptrace` and employs a user-level database to provide transactional updates. Such systems come with significant performance cost (*e.g.,* 50-80% for large operations in DBFS [MTV02]).

In-kernel transactional file systems. An approach that leads to higher performance is adding transactions to in-kernel file systems. Valor [SGC+09] provides

kernel support for file-system transactions. However, Valor does not provide a simple begin/end transaction interface, and it forces programmers to use seven new system calls to manage the transaction log. Valor also adds significant complexity into the kernel.

Microsoft introduced Transactional NTFS (TxF), Transaction Registry (TxR), and the kernel transaction manager (KTM) in Windows Vista [Rus05]. Using TxF requires all transactional operations be explicit (i.e., instead of using `read()` in a transaction, the programmer must add an explicit transactional read). Therefore TxF had a high barrier to entry and code that used it required separate maintenance. TxF also had significant limitations, like no transactions on the root file system.

Transactional operating systems. A third, somewhat heavyweight, approach is modifying the entire operating system to provide transactions. Our prior work, TxOS [PHR+09], is an operating system that provides transactions. This approach adds significant complexity to the kernel. For example, TxOS modified tens of thousands of lines of code and changed core OS data structures like the inode. Maintaining such a kernel is tricky – Windows abandoned its transactional file system and kernel transaction manager [Mic].

The transactional capabilities of the file system supported by TxOS is similar in approach to TxFS. It also uses the file-system journal and modifies the virtual file system (VFS) code to provide isolation. One could view TxFS as specializing TxOS to the file system, achieving a transactional file system at significantly lower cost, while adding file-system specific optimizations like selective journaling and eliminating redundant work within transactions.

Transactional storage systems. Similar to our work, CFS [Min15] provides a lightweight mechanism for atomic updates of multiple files, building on top of transactional flash storage. MARS [Cob13] builds on hardware-provided atomicity to build a transactional system. TxFlash [PRZ08] uses the copy-on-write nature of Flash SSDs to provide transactions at low cost. In contrast to these systems, TxFS provides transactions without assuming any hardware support (beside device cache flush and atomic sector updates). Isotope [Shi16] uses multi-version concurrency control to provide isolation, significantly increasing its complexity. Isotope builds a user-space transactional file system using FUSE, which limits its performance for certain workloads. The higher abstraction level of TxFS makes implementing transactional optimizations and tailored isolation significantly easier than the lower level of Isotope.

Failure atomicity. Failure-atomic msync [PKS13] is similar to TxFS in that it reuses the journal for providing atomicity to application updates; in contrast, TxFS provides full ACID transactions at significantly higher complexity. AdvFS [VMP+15] is also limited in the same way, is specific to the Tru64 file system, and is not available as open-source (latest version available was from 2008). The principles behind TxFS could be used in any file system that has an internal mechanism for atomic updates.

We previewed the ideas behind TxFS at HotOS [HKCW17], and reported on our complete system at ATC [HZN+18]. This paper contains additional experiments and background, along with more details about our design and implementation.

Category	System	Isolation	Durability	Easy-to-use API	Hardware independence	Performance	Complexity
In-kernel transactional FS	**TXFS**	✓	✓	✓	✓	H	L
	Valor	✓	✓	✗	✓	H	L
	TxF	✓	✓	✗	✓	H	H
Transactional OS	TxOS	✓	✓	✓	✓	H	H
FS over userspace databases	OdeFS Inversion DBFS Amino	Relying on databases		✗	✓	L	L
Transactional storage	CFS	✗	✓	✓	✗	H	L
	MARS	✓	✓	✗	✗	H	H
	Isotope	✓	✓	✓	✓	H	H
Failure atomicity	msync	✗	✓	✓	✓	H	L
	AdvFS	✗	✓	✓	✓	H	L

Table 5.1: The table compares prior work providing ACID transactions or failure atomicity in a local file system. Legend: ✓- supported, ✗- unsupported, L - Low, H - High. Note that only TxFS provides isolation and durability with high performance and low implementation complexity without restrictions or hardware modifications.

5.2 Optimizing metadata and data paths

Research [WJX18, VNP+14] shows that after moving from block-based devices to persistent memory, the overhead of the VFS becomes more of an issue. Compared to block-based storage devices, persistent memory has higher throughput in small, random reads/writes. The reduced latency in I/O leads to more observable CPU time. This, along with byte-addressable reads and writes, makes the DRAM

caches less efficient, especially the dentry caches [WJX18]. ByVFS [WJX18] implements a simplified VFS layer with dcache removed. It still keeps the inode cache since it helps achieve higher write throughput. ByVFS does not support concurrency, which is provided by dcache in VFS.

Some previous systems have also explored the idea of more efficient metadata operations. TableFS [RG13] is a FUSE-based file system relying on a user-space LevelDB [Goo] for path name lookup. Although FUSE introduced high system call and kernel-user communication overhead, TableFS still beats Ext4, XFS and Btrfs in many workloads. It does not support kernel bypass on read and write system calls. Since TableFS is FUSE-based, it still relies on the underlying kernel file systems to provide many file-related functionalities. As a result, its scalability, latency and throughput are still restricted by the file systems upon which it builds.

To reduce overhead on context switching and data movement, several file systems choose to bypass the kernel. For example, FLEX and SplitFS bypass the kernel by implementing $read$ and $write$ system calls using $mmap$. Strata [KFH+17] allows the user space library file system to write to an operation log shared with the kernel file system, and relies on the kernel FS to digest the log.

5.3 Persistent data structures

Multiple attempts have been made to develop persistent data structures with crash consistency, such as NV-Tree [YWC+15], wB+ tree [CJ15], FPTree [OLN+16], WOART [LLS+17]. FAST and FAIR [HKWN18], Level Hashing [ZHW18] and CCEH [NCrC+19]. RECIPE [LMK+19] categorizes existing concurrent DRAM

indexes with their ways to guarantee atomicity, and provides a guidance for converting them into persistent indexes with crash consistency.

While prior work focuses on reducing flushes, Zuriel et al. proposed two algorithms on lock-free durable sets [ZFS+19]. Link-free avoids persisting any pointer in the persistent data structures. It maintains in-DRAM data structures with pointers to support fast access to the set nodes, and reconstruct the volatile data structures from the persistent set after a crash. SOFT attempts to further reduce the number of memory fences.

Chapter 6

Conclusion

We present TXFS, a transactional file system built with less development effort than previous systems by leveraging the file-system journal. TXFS is easy to develop, it is easy to use, and it does not have significant overhead for transactions. We show that using TXFS transactions increases performance significantly for a number of different workloads.

Transactional file systems have not been successful for a variety of reasons. TXFS shows that it is possible to avoid the mistakes of the past, and build a transactional file system with low complexity. Given the power and flexibility of file-system transactions, we believe they should be examined again by file-system researchers and developers. Adopting a transactional interface would allow us to borrow decades of research on optimizations from the database community while greatly simplifying the development of crash-consistent applications.

We then propose NVMKVFS, a scalable user-space file system prototype on persistent memory. NVM's low memory access latency makes it possible to eliminate the VFS caches while maintaining performance. NVMKVFS builds its metadata and data indexing from two persistent indexes. The highly parallel metadata and data indexes also handle the concurrent accesses in the file system operations, such as lookups and inode updates. They provide opportunities to implement scal-

able file system calls by removing the scalability bottlenecks introduced by multiple locks in the VFS layer.

Bibliography

[BCB16] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 677694, New York, NY, USA, 2016. Association for Computing Machinery.

[CJ15] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[Cob13] Coburn, Joel and Bunker, Trevor and Schwarz, Meir and Gupta, Rajesh and Swanson, Steven. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[CPADAD13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[Dav] David A. Wheeler. Sloccount. https://www.dwheeler.com/sloccount/.

[DGT15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631644, New York, NY, USA, 2015. Association for Computing Machinery.

[djb] Hash Functions. https://www.cse.yorku.ca/~oz/hash.html.

[fsy] Fsync man page. http://man7.org/linux/man-pages/man2/fdatasync.2.html.

[Geh94] Gehani, Narain H and Jagadish, HV and Roome, William D. OdeFS: A File System Interface to an Object-Oriented Database. In *VLDB*, pages 249–260. Citeseer, 1994.

[GLPT76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[Goo] Google. LevelDB. https://github.com/google/leveldb.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[Hag87] Robert Hagmann. *Reimplementing the Cedar File System Using Logging and Group Commit*, volume 21. ACM, 1987.

[Ham] Hamano. lb - LDAP benchmarking tool like an Apache Bench. https://github.com/hamano/lb.

[HDV+12] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):10, 2012.

[HKCW17] Yige Hu, Younjin Kwon, Vijay Chidambaram, and Emmett Witchel. From Crash Consistency to Transactions. In *16th Workshop on Hot Topics in Operating Systems (HotOS 17)*, Whistler, Canada, 2017.

[HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX*

Winter Technical Conference (USENIX Winter '94), San Francisco, California, January 1994.

[HZN+18] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide ACID transactions. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 879–891. USENIX Association, 2018.

[Int] Intel. *Intel Optane DC Persistent Memory Quick Start Guide*. Intel.

[IYZ+19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv e-prints*, page arXiv:1903.05714, Mar 2019.

[jen] A Hash Function for Hash Table Lookup. `https://www.burtleburtle.net/bob/hash/doobs.html`.

[KFH+17] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[LLS+17] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, February 2017. USENIX Association.

[LMK+19] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 462–477, New York, NY, USA, 2019. ACM.

[MC17] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Systematically Test File-System Crash Consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[Mic] Microsoft. Alternatives to using transactional ntfs. "`https://msdn.microsoft.com/en-us/en-%20us/library/hh802690.aspx`".

[Min15] Min, Changwoo and Kang, Woon-Hak and Kim, Taesoo and Lee, Sang-Won and Eom, Young Ik. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, 2015.

[MKMK16] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[MMP+18] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 33–50. USENIX Association, 2018.

[MTV02] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. `https://goo.gl/3Gj328`, 2002.

[NCrC+19] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[NPM+13] Raghunath Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Andrew Bond, Forrest Carman, and Michael Majdalany. TPC state

of the council 2013. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, volume 8391 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[OBS99] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[Ols93] Michael A. Olson. The design and implementation of the inversion file system. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 205–218. USENIX Association, 1993.

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems*

Design and Implementation (OSDI '14), Broomfield, CO, October 2014.

[PHR+09] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 161–176. ACM, 2009.

[PKS13] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238. ACM, 2013.

[PMC17] Dhathri Purohith, Jayashree Mohan, and Vijay Chidambaram. The dangers and complexities of sqlite benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017*, pages 3:1–3:6. ACM, 2017.

[pmd] Persistent Memory Development Kit. `https://pmem.io/pmdk/`.

[PRZ08] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 147–160. USENIX Association, 2008.

[RG13] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, 2013. USENIX.

[Rus05] Russinovich, Mark E and Solomon, David A and Allchin, Jim. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[SBL+14] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 113–129, 2014.

[SGC+09] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 29–42. USENIX, 2009.

[Shi16] Shin, Ji-Yong and Balakrishnan, Mahesh and Marian, Tudor and Weatherspoon, Hakim. Isotope: Transactional Isolation for Block

Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[SPZ14] Kai Shen, Stan Park, and Men Zhu. Journaling of Journal is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 287–293, 2014.

[SQL] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[Sym] Symas. OpenLDAP. `https://www.openldap.org/`.

[Twe98] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[VMP+15] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya S Mannarswamy, Terence Kelly, and Charles B Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 203–211, 2015.

[VNP+14] Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[WJX18] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[Wri07] Wright, Charles P and Spillane, Richard and Sivathanu, Gopalan and Zadok, Erez. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.

[XS16] Jian Xu and Steven Swanson. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[YWC+15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[ZFS+19] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA):128:1–128:26, October 2019.

[ZHW18] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 461–476, Carlsbad, CA, October 2018. USENIX Association.

Vita

Yige Hu was born in Lanzhou, China and raised in Hangzhou, China. In 2013, she graduated from both Tongji University with a B.E. degree in Electronic and Information Engineering and Polytechnic University of Turin with a B.S. degree in Computer Engineering. In August 2013, she entered the doctoral program in the Department of Computer Science at the University of Texas at Austin, where she received an M.S. degree in Computer Science in May 2020.

Permanent address: yige@cs.utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.